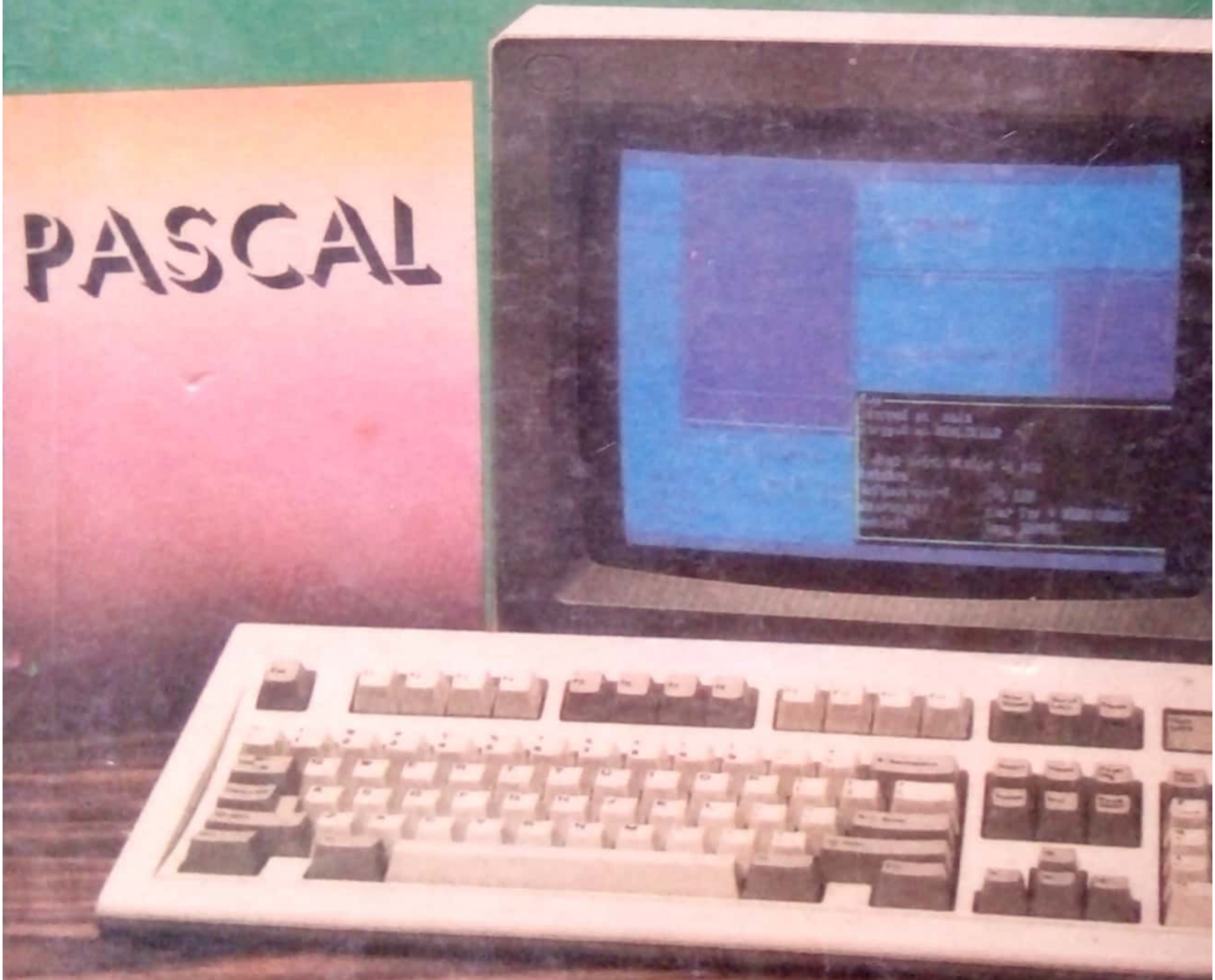
POPULAR APPLICATIONS SERIES

Companion Disk Available

LEARN Pascal IN THREE DAYS



SAM ABOLROUS

Learn Pascal in Three Days

Sam A. Abolrous

BPB PUBLICATIONS
B-14, CONNAUGHT PLACE, NEW DELHI-110001

FIRST INDIAN EDITION - 1994

Distributors:

BPB BOOK CENTRE
376, Old Lajpat Rai Market, DELHI-110006

BUSINESS PROMOTION BUREAU
8/1, Ritchie Street, Mount Road, MADRAS-600002

BUSINESS PROMOTION BUREAU
4-3-268-C, Girlraj Lane, Bank Street, HYDERABAD-500195

COMPUTER BOOK CENTRE
12, Shrungar Complex, M. G. Road, BANGALORE-560001

COMPUTER BOOK CENTRE
SCF No.-65, Sector-6, PANCHKULA-134109, CHANDIGARH

MS-DOS and Xenix are registered trademarks of Microsoft Corporation.
PC-DOS is a registered trademark of International Business Machines Corporation.
Turbo Pascal is a registered trademark of Borland International.

Other product names mentioned are used for identification purposes only and may be trademarks of their respective companies.

Printed in India under arrangement with
WORDWARE PUBLISHING INC., USA

ISBN 81-7029-268-9

Published by Manish Jain for BPB Publications, B-14, Connaught Place, New Delhi and Printed at NU TECH PACKAGINGS, B-56 B, Sector VII, NOIDA (U.P.)

Contents

DAY TWO

DAY THREE

PREFACE

"Is it possible to **Learn Pascal in Three Days?**" The answer is YES. If the jargon of the language is avoided, you can take a shortcut straight to its heart.

The method used in this book is that of learning by example. You start with simple programs which crunch some numbers and print some strings, and you end up with useful applications using structured data and files.

Historically, Pascal was developed by Niklaus Wirth (a Swiss computer scientist) in the early 1970s and was named after the French mathematician Blaise Pascal (1623-1662). A recent standard for the language was formulated in 1983 and approved by the Institute of Electrical and Electronic Engineers (IEEE) and the American National Standards Institute (ANSI). With the growing use of microcomputers, extensions and variations have been added to the language, the most popular of which are UCSD Pascal (developed by University of California at San Diego) and Turbo Pascal (developed by Borland International). The goal of this book is to teach you how to write a portable program in Pascal regardless of the computer you use, so the standard IEEE/ANSI will mainly be used but the new features will be discussed and their origins referred to. The book is not intended to go into the details of the nonportable areas of the language (such as high resolution graphics), but rather to make use of the powerful features of the modern implementations (especially Turbo Pascal) that help in data processing. The programs included in this book were compiled using Turbo Pascal, but you can use any compiler to run them. In a very few places you may need to make minor modifications, which will be referenced.

Sam A. Abolrous

DAY ONE

CHAPTER ONE

HELLO Pascal

1-1 YOUR FIRST PASCAL PROGRAM

The Pascal program may be as simple as the one in figure 1-1. It displays on your screen the phrase "Hi there."

```
{----------------------------- figure 1-1 ------------------------------}
PROGRAM FirstProgram(OUTPUT);
BEGIN
   WRITELN('Hi there')
END.
```

Whether the Pascal program is small or large, it must have a specific structure. This program consists mainly of one statement (WRITELN) which does the actual work here, as it displays whatever comes between the parentheses. The statement is included inside a frame starting with the keyword BEGIN and ending with the keyword END. This is called the *program main body* (or the program block) and usually contains the main logic of data processing.

COMMENTS Consider the first line in the program:

```
{----------------------------- figure 1-1 ------------------------------}
```

This is a *comment* and is totally ignored by the compiler. Comments can appear anywhere in the Pascal program between two braces ({ }) or between the two symbols "(*" and "*)" thus:

```
(* This is a comment *)
```

PROGRAM HEADING The second line is called the *program heading*. It starts with the keyword PROGRAM followed by a space, followed by the program name (FirstProgram). The program name is a user-invented word. User-invented words are classified in Pascal as *identifiers*. An identifier must

begin with a letter and may contain any number of letters or digits (in Turbo Pascal it may contain underscores as well). You are free to choose any meaningful name for your program, but do not expect a program name like "BEGIN" or "PROGRAM" to be accepted. These words are called *reserved words*, and they are only used in the proper place in the program. Pascal reserved words are summarized in Appendix B.

The program name is followed by the word OUTPUT contained in parentheses and terminated with a semicolon:

```
PROGRAM FirstProgram(OUTPUT);
```

The keyword OUTPUT tells the compiler that this program is going to produce output (such as writing to the screen), which is the counterpart of INPUT (such as reading from the keyboard). The words OUTPUT and INPUT are called *file parameters*. The program may perform both input and output, in which case the file parameters take the form:

```
PROGRAM FirstProgram(INPUT,OUTPUT);
```

In Turbo Pascal the program heading is optional. You may skip the whole line and start your program with the word BEGIN, or you may use the program name without parameters, like this:

```
PROGRAM FirstProgram;
```

SYNTAX AND CONVENTIONS The most important syntax is the semicolon after the program heading (which is used as a separator) and the period after the word END (which terminates the program).

A common convention is to write Pascal keywords in uppercase and the user-invented names (identifiers) in lowercase with the first letter capitalized. If the name consists of more than one word (which is the case in this program), the first letter in each word is capitalized. So, in Pascal programs you may see identifiers like:

Wages
PayRoll
HoursWorkedPerWeek

This is just a convention to make your program readable, but Pascal compilers are not case sensitive. This means that you can write the entire program in lowercase as in figure 1-2, or in uppercase as in figure 1-3. All three of the programs will compile and run.

```
{ ----------------------------- figure 1-2 ------------------------------ }
program firstprogram(output);
begin
  writeln('Hi there')
end.
```

```
{ ----------------------------- figure 1-3 ------------------------------ }
PROGRAM FIRSTPROGRAM(OUTPUT);
BEGIN
  WRITELN('Hi there')
END.
```

All blank lines, indentation, and spaces (except those following the Pascal keywords) are optional, but it is a good programming habit to use this method to make your program well-organized and readable.

1-2 DISPLAYING TEXT: WRITELN, WRITE

To display several lines of text you need a WRITELN statement for each line, as in the following program in figure 1-4.

NOTE A companion diskette comes with this book and contains the source code of all examples, in addition to the solutions of the drills. Please read the "readme" file on the distribution disk. Just type the command README and press Enter.

```
{ ----------------------------- figure 1-4 ----------------------------- }
PROGRAM LinesOfText(OUTPUT);
BEGIN
  WRITELN('Hi there.');
  WRITELN('How are you today?');
  WRITELN('Are you ready for Pascal?')
END.
```

Now the program contains more than one statement. Each statement must be separated from the next one with a semicolon. This is the only way the compiler can recognize the end of a statement, but for the last statement in the program block you may skip the semicolon.

When you compile this program it will give the following output:

```
Hi there.
How are you today?
Are you ready for Pascal?
```

The WRITELN statement displays a line of text followed by a new line (a line feed and a carriage return). If you wish to display two strings on the same line, you need to use the statement WRITE as shown in the following program.

```
{ ------------------------------ figure 1-5 ------------------------------ }
PROGRAM TwoLines(OUTPUT);
BEGIN
   WRITE('Hi there. ');
   WRITELN('How are you today?');
   WRITELN('Are you ready for Pascal?')
END.
```

The output of this program is:

```
Hi there. How are you today?
Are you ready for Pascal?
```

As you can see in the program output, the second string is written on the same line as the first string as a result of using the statement WRITE to display the first string. This is the only difference between the two output statements WRITE and WRITELN.

If you want to display a blank line, you only need the statement:

```
WRITELN;
```

DRILL 1-1

Write a Pascal program to display the following text on the screen:

```
WORDWARE PUBLISHING, INC.
-------------------------
1506 CAPITAL AVENUE
PLANO, TEXAS 75074
```

1-3 CRUNCHING NUMBERS

The easiest task for any program is to crunch numbers. The statement WRITELN (or WRITE) can be used both to display numbers and evaluate numerical expressions. You can build up arithmetic expressions using the following *arithmetic operators*:

+	for addition	-	for subtraction
*	for multiplication	/	for division

Take a look at these examples:

```
WRITELN(123);
WRITELN(1.23 * 4);
```

The first example displays the number that came between the parentheses (123). The second example performs multiplication of two numbers and displays the result. Notice that for numeric values, unlike text strings, you don't use quotes.

You may use WRITELN to display text and numbers in the same statement by using the comma as a separator like this:

```
WRITELN('The result is=', 125 * 1.75);
```

The following program is used to evaluate two numeric expressions (multiplication and division) and display the results preceded by the proper text.

```
{ ------------------------------ figure 1-6 ------------------------------}
PROGRAM CrunchNumbers(OUTPUT);
BEGIN
  WRITELN('I can easily crunch numbers.');
  WRITELN('Here is multiplication of 50x4:',50*4);
  WRITELN('..and here is division of 2400/8:',2400/8)
END.
```

The output of this program is:

```
I can easily crunch numbers.
Here is multiplication of 50x4:200
..and here is division of 2400/8: 3.0000000000E+02
```

The multiplication is done as expected. The two operands (50 and 4) were integers (whole numbers) and the result (200) was an integer too. The division result, however, came out in a format that needs some explanation.

INTEGERS AND REAL NUMBERS The division performed with the operator / is called *real division* and always produces as its result a real number. Real numbers may be written in fixed-point notation (such as 300.0) or in scientific (exponential) notation (such as 3.0E+02), but in Pascal, real number output will always be represented in scientific notation by default. A number written in scientific notation is made up of two parts divided by the letter E (or e). The left part is called the *mantissa* and indicates the significant digits, while

the right part is called the *exponent*. The exponent is a power of ten that determines the position of the decimal point. So, in this example the number:

3.0000000000E+02

is the same as the number:

3×10^2

The same number, when expressed in fixed-point format, becomes:

300.0

If the exponent is preceded by a minus sign as in:

3.124E–02

then the decimal point is shifted two positions to the left. This number, then, is the same as:

0.03124

If the number is negative, the minus sign should precede the mantissa:

–0.0124E–02

If the number is positive, you may omit the sign for either the mantissa or the exponent:

1.23E02

The division operator (/) is called the *real division operator*, because the result always appears as a real number regardless of the type of the operands.

For *integer division* use the operator DIV as in the example:

```
WRITELN(2400 DIV 8);
```

This will produce the output 300.

With integer division, any fraction in the result will be truncated, as in this example:

```
WRITELN(9 DIV 4);    produces the output 2.
```

Another important operator, MOD, is used to get the remainder of integer division (Modulo), as in these examples:

```
WRITELN(9 MOD 4);    produces the output 1.
WRITELN(3 MOD 4);    produces the output 3.
```

The operators DIV and MOD take only integer operands and produce integer output.

For the other operators (+, –, and *), if either one of the operands is real, the result will be real.

DRILL 1-2

Evaluate the following expressions and write the result either as an integer (if integer), or as a fixed-point real number (if real):

A. 144 / 12

B. 144 DIV 12

C. 17 MOD 5

D. 3 MOD 5

E. 3e+02 + 3

F. 345E–01 – 1

EVALUATION OF ARITHMETIC EXPRESSIONS When you build more complicated arithmetic expressions, you have to watch the priority of each operator involved in the expression. Take a look at these two expressions:

```
2 + 10 / 2
(2 + 10) / 2
```

Although the two expressions use the same numbers and operators, the first expression is evaluated as 7, while the second is evaluated as 6. This is because in the first expression the division is evaluated before the addition, while in the second expression the parentheses are used to change the order of evaluation, in which case the expression inside the parentheses is evaluated first. In general, the arithmetic operators in Pascal have two levels of precedence: *high* and *low*.

The + and – have low precedence, while all other operators have high precedence.

If an expression contains two operators of the same precedence level, they are evaluated from left to right. Consider this example:

```
5 + 3 * 2 - 6 DIV 2
```

The first operation to be performed is the multiplication:

```
5 + 6 - 6 DIV 2
```

The second operation, of next highest priority, is the division:

```
5 + 6 - 3
```

This leaves two operations of equal priority. They are evaluated from left to right giving:

```
8
```

When parentheses are used to alter the order of evaluation, they form subexpressions which are evaluated first. If parentheses are nested, the innermost subexpressions are evaluated first. Consider the same example with nested parentheses:

```
((5 + 3) * 2 - 6) DIV 2
```

This expression is evaluated according to the following steps:

```
(8 * 2 - 6) DIV 2
(16 - 6) DIV 2
10 DIV 2
5
```

Arithmetic operators are summarized in table 1-1, along with their precedence and properties.

The + and – signs are also used as *unary operators* (to signify positive and negative). The unary operators are of the same low priority as the *binary operators* + and –. If a binary operator precedes the unary operator such as 5 * – 4, you must enclose the unary operator and its number in parentheses: 5 * (–4). The first form may be accepted by some compilers, but do not try it.

Table 1-1 Arithmetic Operators

Operator	*Arithmetic Operation*	*Operands*	*Result*	*Precedence*
+	Addition	REAL/INTEGER	REAL/INTEGER	Low
–	Subtraction	REAL/INTEGER	REAL/INTEGER	Low
*	Multiplication	REAL/INTEGER	REAL/INTEGER	High
/	Real division	REAL/INTEGER	REAL	High
DIV	Integer division	INTEGER	INTEGER	High
MOD	Remainder of integer division	INTEGER	INTEGER	High

DRILL 1-3

Evaluate the following expressions and write the result either as an integer (if integer), or as a fixed-point real number (if real):

A. 15 – 15 DIV 15

B. 22 + 10 / 2

C. (22 + 10) / 2

D. 50 * 10 – 4 MOD 3 * 5 + 80

1-4 USING VARIABLES

Data are stored in the memory locations at specific addresses. Programmers, however, refer to these locations using variables. When variables are used in a program, they are associated with the specific memory locations. The value of a variable is actually the contents of its memory location. As data are processed by the program, the contents of any location may change, and so does the value of the associated variable. Variables are given names (identifiers) according to the rules mentioned before.

VARIABLE DECLARATION Before using a variable in a Pascal program, its name and *type* must be declared in a special part of the program called the *declaration part*. This part starts with the keyword VAR, as in the following example:

```
VAR
   a :INTEGER;
   x :REAL;
```

The variable "a" is of the type INTEGER, which means that it can hold only integer numbers such as 4, 556, and 32145. The variable "x" is declared as of the type REAL and can hold real numbers such as 3.14, 44.567, and 3.5E+02.

If you want to declare more than one variable of the same type, you may declare each on a separate line:

```
VAR
   a :INTEGER;
   b :INTEGER;
   c :INTEGER;
   x :REAL;
   y :REAL;
```

or, you may also declare all variables of the same type as a list like this:

```
VAR
   a, b, c:INTEGER;
   x, y   :REAL;
```

The keywords INTEGER and REAL are classified as *standard identifiers*, which are predefined in Pascal. The standard identifiers can be redefined by the programmer, but this is strongly recommended against. Standard identifiers are listed in Appendix B.

In the following program three variables are declared: "a" and "b" are integers, while "x" is real. The contents of each one are displayed using the WRITELN statement.

```
{ ------------------------------ figure 1-7 ---------------------------- }
PROGRAM Variables(OUTPUT);
{ Variable Declarations }
VAR
   a, b :INTEGER;
   x    :REAL;
{ Program Block }
BEGIN
   WRITELN('Contents of a=',a);
   WRITELN('Contents of b=',b);
   WRITELN('Contents of x=',x)
END.
```

The output of the program is something like the following:

```
Contents of a=0
Contents of b=631
Contents of x= 2.7216107254E-26
```

Note that the contents of "a" and "b" are displayed as integers while the contents of "x" are displayed in real format. However, the output numbers are just garbage because no values were actually stored in those variables. Unless you store data values in your variables, they will contain whatever was last left in those memory locations.

THE ASSIGNMENT STATEMENT To store a value in a variable you can use the *assignment operator* (:=) as in the following examples:

```
a := 55;
x := 1.5;
y := 2.3E+02;
```

CAUTION

Do not use a real number like this:

```
.1234
```

A legal real number in Pascal must have a digit to the left of the decimal point, like this:

```
0.1234
```

Also, the number:

```
123.
```

may be rejected by some compilers. It would be better to use the legal form:

```
123.0
```

In the following program, two integers "a" and "b" are declared in the declaration part, then assigned integer values in the program block. The WRITELN statement is then used to evaluate and display the results of different arithmetic operations performed on those variables.

```
{ ------------------------------ figure 1-8 ------------------------------ }
PROGRAM Arithmetic(OUTPUT);
{ Variable Declarations }
VAR
   a, b :INTEGER;
{ Program block }
BEGIN
   a := 25;
   b := 2;
   WRITELN('a=',a);
   WRITELN('b=',b);
   WRITELN('a+b=',a+b);
   WRITELN('a-b=',a-b);
   WRITELN('a*b=',a*b);
   WRITELN('a/b=',a/b);
   WRITELN('a div b=',a DIV b);        { used with integers only }
   WRITELN('a mod b=',a MOD b)         { used with integers only }
END.
```

The output of the program is:

```
a=25
b=2
a+b=27
a-b=23
a*b=50
a/b= 1.2500000000E+01       ----> Real division
```

```
a div b=12          ----> Integer division
a mod b=1           ----> Remainder of integer division
```

You may assign one variable to another thus:

```
x := y;
```

In this case, the contents of the variable "y" are copied to the variable "x." You may also assign an arithmetic expression to a variable, like this:

```
z := a + b - 2;
GrossPay := PayRate * HoursWorked;
```

In these statements the value of the expression to the right of the assignment operator is calculated and stored in the variable to the left of the assignment operator ("z" or "GrossPay").

DRILL 1-4

Write a Pascal program to do the following:

A. Assign the value "2" to a variable "a," and the value "9" to a variable "b."

B. Display the values of the expressions:

```
a+b DIV 2
(a+b) DIV 2
```

1-5 NAMED CONSTANTS

Data values (in many languages including Pascal) are called constants, as they never change during the program execution. In Pascal there are two types of constants:

- Literal constants
- Named constants

Literal constants are data values such as explicit numbers and text strings, while a named constant is a "constant variable." The difference between a named constant and a variable is that the value of the named constant does not change during the program. Like variables, a named constant is given a name and has to be declared in the declaration part. Actually, the declaration part is divided into two sections, CONST and VAR; the CONST section comes before the VAR section. Suppose that you would like to use the value 3.14159 (a numerical constant known as "PI") many times in your calculations. It would

be more convenient to give it a name and use the name in your code. You can declare named constants as in the following example:

```
CONST
   Pi = 3.14159;
   ThisYear = 1992;
   Department= 'OtoRhinoLaryngology';
```

Some constants are predefined in Pascal as standard identifiers. One useful predefined named constant is MAXINT, which gives the maximum value an integer can possess. The value depends on the computer used. If you want to know the value of MAXINT in your computer, use the statement:

```
WRITELN(MAXINT);
```

A typical value is 32767 (two bytes).

In the following program, the perimeter of a circle is calculated using the named constant Pi.

```
{ ------------------------------ figure 1-9 ---------------------------- }
PROGRAM Constants(OUTPUT);
{ Constant Declarations }
CONST
   Pi = 3.14159;
{ Variable Declarations }
VAR
   Radius, Perimeter  :REAL;
{ Program block }
BEGIN
   Radius := 4.9;
   Perimeter := 2 * Pi * Radius;
   WRITELN('Perimeter=', Perimeter)
END.
```

The output of this program is:

```
Perimeter= 3.0787582000E+01
```

NOTE If you are using Turbo Pascal, you do not need to redefine the constant Pi, as it is predefined as a standard identifier.

1-6 TYPE CONVERSION: ROUND, TRUNC

You can assign an integer to a variable of the type REAL, but the opposite is not permitted. The reason for this is because the storage size allocated for an integer is smaller than that allocated for a real number. If this were permitted, data could be lost or corrupted when a large number was moved to a smaller location in which it did not fit. You can, however, perform the conversion with one of the two functions:

ROUND(n) rounds "n" to the closest integer
TRUNC(n) truncates the fraction part of "n."
where:
n is a real variable or expression.

Consider these examples:

```
ROUND(8.4)      returns 8
ROUND(8.5)      returns 9
TRUNC(8.4)      returns 8
TRUNC(8.5)      returns 8
```

As you can see in the examples, the two functions may or may not return the same integer value for the same argument.

In the following program the two functions are used to get the rounded and the truncated integer values of the real variable "Perimeter."

```
{ ----------------------------- figure 1-10 ----------------------------- }
PROGRAM Functions1(OUTPUT);
{ Constant Declarations }
CONST
   Pi = 3.14159;
{ Variable Declarations }
VAR
   Perimeter, Radius                          :REAL;
   RoundedPerimeter, TruncatedPerimeter       :INTEGER;
{ Program block }
BEGIN
   Radius := 4.9;
   Perimeter := 2*Pi*Radius;
   RoundedPerimeter := ROUND(Perimeter);
   TruncatedPerimeter := TRUNC(Perimeter);
   WRITELN('Perimeter=', Perimeter);
```

```
   WRITELN('Perimeter (rounded)=', RoundedPerimeter);
   WRITELN('Perimeter (truncated)=', TruncatedPerimeter)
END.
```

The output is:

```
Perimeter= 3.0772000000E+01        ---->   The actual result
Perimeter (rounded)=31             ---->   Rounded result
Perimeter (truncated)=30           ---->   Truncated result
```

1-7 READING FROM THE KEYBOARD: READLN, READ

The previous program is used to calculate the perimeter for a given radius, hard coded in the program. A more useful program would accept the radius from the user, do the calculations, then display the result. You can use either READLN or READ to make the program pause and wait for user input. The READLN statement is used to read the value of one or more variables. It takes the general form:

READLN(variable-list);

To read the value of a variable "x" from the keyboard, you can use the statement:

```
READLN(x);
```

To read the values of three variables x, y, and z, use the statement:

```
READLN(x, y, z);
```

When you enter the values of more than one variable (such as x, y, and z), they should be separated by one or more blanks or by pressing the **Enter** key.

Replace the assignment statement in the previous program with a READLN statement as follows:

```
READLN(Radius);
```

If you try the program now, it will pause until you type a number and press **Enter**; it then resumes execution and displays the results. Unfortunately, you cannot use the READLN statement to display a user prompt when the program is waiting for input. This must be done using a WRITE (or WRITELN) statement such as:

```
WRITE('Please enter the radius:');
```

Here is the program in its new shape:

```
{ ------------------------------ figure 1-11 ---------------------------- }
PROGRAM KeyboardInput(OUTPUT);
{ Constant Declarations }
CONST
   Pi = 3.14159;
{ Variable Declarations }
VAR
   Perimeter, Radius                     :REAL;
   RoundedPerimeter, TruncatedPerimeter  :INTEGER;
{ Program block }
BEGIN
   WRITE('Please enter the radius:');
   READLN(Radius);
   Perimeter := 2*Pi*Radius;
   RoundedPerimeter := ROUND(Perimeter);
   TruncatedPerimeter := TRUNC(Perimeter);
   WRITELN('Perimeter=', Perimeter);
   WRITELN('Perimeter (rounded)=', RoundedPerimeter);
   WRITELN('Perimeter (truncated)=', TruncatedPerimeter)
END.
```

A sample run of the program gives the following:

```
Please enter the radius:4.9         ----> Type the number and press Enter
Perimeter= 3.0787582000E+01
Perimeter (rounded)=31
Perimeter (truncated)=30
```

NOTE At this stage you can use either READ or READLN for keyboard input as the difference between them is not noticeable in our applications so far.

1-8 FORMATTING OUTPUT

You have probably thought that scientific notation is not the best format for output, especially with business and money figures. You bet. Scientific notation is useful only with very large or very small numbers, where the power of ten represents an order of magnitude of the number.

Whenever you want to see your results in fixed-point notation, use the *format descriptors* as in this example:

```
WRITELN(Wages :6:2);
```

The format ":6:2" determines a field width of 6 positions, including 2 decimal places. So, if the value of the variable "wages" is 45.5 it will be displayed as:

```
B45.50
```

where the letter "B" refers to a blank space. If the output digits are less than the field width, which is the case in this example, the result will be right shifted. If the number is larger than the field width, then the field will be automatically enlarged and the entire number printed.

You can add a character (such as the dollar sign) to the left of the number as follows:

```
WRITELN('$',Wages :6:2);
```

This will produce the output:

```
$ 45.50
```

By using a smaller field width, you can have the number shifted to the left and the dollar sign attached to the first significant digit:

```
WRITELN('$',Wages :0:2);
```

This will produce:

```
$45.50
```

You can format any type of data using the same method. The only difference is that with integers or strings you specify the width field without decimal places.

In the following program different types of data are formatted to fit into specific fields, as shown in the output.

```
{ ---------------------------- figure 1-12 ---------------------------- }
PROGRAM Format(OUTPUT);
{ Variable Declarations }
VAR
  a     :INTEGER;
  b     :REAL;
{ Program Block }
BEGIN
  b := 1.2e+02;
  a := 320;
  WRITELN('I am a text string starting from position 1.');
  WRITELN('I am now shifted to the right end of the field.':50);
```

```
  WRITELN('I am an unformatted integer:', a);
  WRITELN('I am an integer written in a field 6 characters wide:', a:6);
  WRITELN('I am a money amount written in 8 positions:$',b:8:2);
  WRITELN('I am a money amount shifted to the left:$',b:0:2)
END.
```

The output is:

```
I am a text string starting from position 1.
    I am now shifted to the right end of the field.
I am an unformatted integer:320
I am an integer written in a field 6 characters wide:   320
I am a money amount written in 8 positions:$  120.00
I am a money amount shifted to the left:$120.00
```

If you display the numeric variables alone (without text), they will appear as follows:

```
320
   320
$  120.00
$120.00
```

DRILL 1-5

Write a program to calculate employee wages according to the formula:

```
Wages := HoursWorked * PayRate;
```

Accept the "HoursWorked" and the "PayRate" from the keyboard and display the "Wages" in fixed-point notation preceded by a dollar sign.

SUMMARY

In this chapter you were introduced to the most important tools in Pascal programming.

1. You are now familiar with the Pascal program structure:
   - The program heading
   - The Declaration part
     - The **CONST** section
     - The **VAR** section
   - The program main body between **BEGIN** and **END**.

2. You know two important data types, **INTEGER** and **REAL**, and how to express and evaluate arithmetic expressions using both types.
3. You know the arithmetic operators in Pascal, their properties, and their precedence.

   \+   –   *   /   **DIV**   **MOD**

4. You know how to declare variables of both types, how to name them using identifiers, how to store values in them whether by assignment (:=) or by entering values from the keyboard, and how to display their values on the screen.
5. You know as well how to declare named constants and use them in the program.
6. During your first tour of Pascal, you learned the following output statements to display both variables and numeric or string literal constants:

   WRITELN
   WRITE

   Also, you learned the following input statements to read variable values from the keyboard:

   READLN
   READ

7. Finally, you learned how to format your numeric or string output to have the results in the desired form.

CHAPTER TWO

LANGUAGE ELEMENTS

2-1 STANDARD DATA TYPES AND FUNCTIONS

The data processed by any program may consist of integers, real numbers, or strings of text, but each type is stored and manipulated differently. Pascal provides the following standard data types (also referred to as simple or *scaler* data types):

INTEGER
REAL
CHAR
BOOLEAN

You have already used the INTEGER and REAL types as both numeric constants and variables. You have also already used arithmetic operators with variables and constants to build arithmetic expressions, and you tasted the flavor of some functions such as ROUND and TRUNC. This chapter introduces the whole picture of numeric data types and related functions and expressions. It also introduces the type CHAR which is used to represent single characters, and the type BOOLEAN to represent logical values. The discussion of the single character type contains an overview of how strings were represented in standard Pascal and also how they are represented in the modern implementations such as Turbo Pascal and UCSD Pascal (using the type STRING).

2-2 NUMERIC DATA TYPES

The range of numbers that may be represented as integers (or as reals) depends on the implementation. For the type INTEGER it is determined by the following limits:

MAXINT	the maximum positive integer, and
–(MAXINT+1)	the maximum negative integer.

Again, the value of MAXINT depends on the implementation.

Real numbers are generally stored in a larger number of bytes than are integers, but they are of limited precision. Fractions such as 0.333333 and 0.666666 can never be as precise as the exact values 1/3 and 2/3, regardless of how many digits are used to represent the number. For this reason, it is not recommended to test two real numbers for equality. Instead, it would be better to test to see if the difference between the two numbers is less than some specific small amount.

In Turbo Pascal, there are additional numeric types, which are introduced in the following section.

NUMERIC TYPES IN TURBO PASCAL There are additional integer types (including the type INTEGER) in Turbo Pascal. They are shown in table 2-1 along with their storage sizes and the maximum range of values that can be represented in each.

In one byte, you can store either a SHORTINT or a BYTE. The BYTE is actually an unsigned SHORTINT, which means that it can hold only positive numbers. As you can see in the table, the maximum range of values for a type is doubled when the sign is not used. The same applies to the types INTEGER and WORD, as the WORD is a positive integer of doubled maximum range.

Table 2-1 Turbo Pascal Integer Types

Data Type	*Size (in bytes)*	*Range*
SHORTINT	1	from –128 to +127
BYTE	1	from 0 to 255
INTEGER	2	from –32768 to +32767
WORD	2	from 0 to 65535
LONGINT	4	from –2,147,483,648 to +2,147,483,647

The LONGINT is the largest integer that can be represented in Turbo Pascal. You can test its value by displaying the value of the predefined constant MAXLONGINT as follows:

```
WRITELN(MAXLONGINT);
```

Notice that the negative range of any signed type exceeds the positive range by one (e.g. +127 and –128). This is because zero is counted with the positive numbers.

CAUTION

The commas used here to express large numbers are used only for readability. You will neither see them in the output of a program, nor are they accepted as a part of literal constants. So, the number 2,147,483,647 must be used as 2147483647.

In Turbo Pascal, there are also additional real types (including the type REAL) as shown in table 2-2. For real numbers, a new column is added to the table to describe the accuracy of a number as the maximum number of precise digits.

Table 2-2 Turbo Pascal Real Types

Data Type	*size (in bytes)*	*Precision (up to)*	*Range*
SINGLE	4	7 digits	from 0.7–45 to 3.4E+38
REAL	6	11 digits	from 2.94E–39 to 1.7E+38
DOUBLE	8	15 digits	from 4.94E–324 to 1.79E+308
EXTENDED	10	19 digits	from 3.3E–4932 to 1.18E+4932
COMP	8	integers only	9.2E+18

If you examine the range of the type SINGLE, you will find that it is pretty close to that of the type REAL, especially in the area of the very large numbers. The main difference between the two lies in the economical storage of the SINGLE type (4 bytes compared to 6), which comes at the expense of precision (7 digits compared to 11). Real number types other than REAL are not available unless a math coprocessor is used. The type COMP actually belongs to the set of integers, as it does not accept fractions, but it is usually mentioned among reals as it requires the use of a math coprocessor.

2-3 STANDARD ARITHMETIC FUNCTIONS

Pascal includes a large number of predefined functions that may be used in expressions among constants and variables. Table 2-3 shows the standard arithmetic functions divided into three groups:

A. The conversion functions

B. The trigonometric functions

C. Miscellaneous functions

Any function operates on a parameter that comes inside its parentheses. The parameter is an expression of a specific type (notice that the expression may be a single variable or constant). Before using any of these functions, you must know the type of parameter the function uses and the type of the returned value (which is also the type of the function). The conversion functions, for instance, take real parameters and return integer results. Other functions use either integer or real parameters, and produce different types. The type of the returned value is important when you assign the function to a variable.

Table 2-3 Standard Arithmetic Functions

Function format	*Returned value*	*Parameter type*	*Result type*
Conversion functions:			
ROUND(x)	x rounded to the nearest integer	REAL	INTEGER
TRUNC(x)	x with the fraction part truncated	REAL	INTEGER
**Trigonometric functions:*			
ARCTAN(x)	The arctangent of x	REAL/INTEGER	REAL
COS(x)	Cosine of x	REAL/INTEGER	REAL
SIN(x)	Sine of x	REAL/INTEGER	REAL
Miscellaneous functions:			
ABS(x)	The absolute value of x	REAL/INTEGER	REAL/INTEGER
EXP(x)	The exponential function of x (e^x)	REAL/INTEGER	REAL
LN(x)	The natural logarithm of x	REAL/INTEGER	REAL
SQR(x)	The square of x (x)	REAL/INTEGER	REAL/INTEGER
SQRT(x)	The square root of x (x)	REAL/INTEGER	REAL

* All angles must be expressed in radians.

Look at these examples:

SQR(3)=9
SQR(2.5)=6.25
SQRT(9)=3.00
ABS(-28.55)=28.55
LN(EXP(1))=1.00
ARCTAN(1)=45 degrees (see the note below)

Note that the type of result returned by the function SQR is the same as the type of the parameter, but the function SQRT returns a real number regardless

of the parameter type. Notice also that the parameter of any function may contain another function, such as LN(EXP(1)).

The output returned from the last function (ARCTAN) is here converted to degrees but will come in radians if not converted. The program which produced these results is shown in figure 2-1. Pay attention to the format descriptors, which are used to produce the output in these formats.

```
{ ---------------------------- figure 2-1 ------------------------------ }
{ Arithmetic Standard Functions }
PROGRAM FunctionDemo(OUTPUT);
CONST
   Pi = 3.14159;                  { No need for this part in Turbo Pascal }
BEGIN
   WRITELN('SQR(3)=',SQR(3));
   WRITELN('SQR(2.5)=',SQR(2.5):0:2);           { Notice the format }
   WRITELN('SQRT(9)=',SQRT(9):0:2);
   WRITELN('ABS(-28.55)=',ABS(-28.55):0:2);
   WRITELN('LN(EXP(1))=',LN(EXP(1)):0:2);
   WRITELN('ARCTAN(1)=',ARCTAN(1)* 180/Pi:0:0,' degrees')
                          { Notice the conversion and the format }
END.
```

EXAMPLE: THE POWER FUNCTION The power operator does not exist in Pascal as it does in some other languages (such as FORTRAN and BASIC), but you can make one using arithmetic functions. You can, of course, use the function SQR to produce small powers, thus:

SQR(x)	power 2
SQR(x) * x	power 3
SQR(SQR(x))	power 4

You can also make use of the following mathematical relationship to express any power:

$$x^y = EXP(LN(x) * y)$$

In the following program this expression is used to raise a number to any power. The program asks you to enter both the base "x" and the exponent "y," then displays the formatted result.

```
{ ---------------------------- figure 2-2 ----------------------------}
{ Arithmetic Standard Functions }
PROGRAM PowerOperator(INPUT,OUTPUT);
VAR
```

```
  a, b :REAL;
BEGIN
  WRITE('Enter the base and the exponent separated by a space:');
  READLN(a,b);
  WRITELN('The value of ',a:0:2,' raised to the power ',b:0:2,' is ',
          EXP(LN(a)*b):0:2)
END.
```

A sample run of the program gives the following:

```
Enter the base and the exponent separated by a space:2 10
The value of 2.00 raised to the power 10.00 is 1024.00
```

EXAMPLE: GROCERY STORE In a grocery store a fast calculation is needed to count the number and type of coins that make up the change remaining from a dollar, so it is a great help to have this logic programmed into the cash register. The following program accepts from the keyboard the price of the purchase (for the sake of simplicity, this is assumed to be less than one dollar) and produces as output the number of quarters, dimes, and nickels remaining from a dollar bill. The program is an application of the integer operators DIV and MOD.

```
{ ---------------------------- figure 2-3 ----------------------------- }
{ Grocery Store }
PROGRAM Grocery(INPUT,OUTPUT);
VAR
  Change, TotalPrice,
  Dollars, Quarters, Dimes, Nickels, Cents :INTEGER;
BEGIN
  WRITE('Enter the total-price in cents: ');
  READLN(TotalPrice);
  Change := 100 - TotalPrice;
  { Quarters }
    Quarters := Change DIV 25;
    Change := Change MOD 25;
  { Dimes }
    Dimes := Change DIV 10;
    Change := Change MOD 10;
  { Nickels }
    Nickels := Change DIV 5;
    Change := Change MOD 5;
  { Cents }
    Cents := Change;
  WRITELN('The change is:');
```

```
   WRITELN(Quarters,' Quarters');
   WRITELN(Dimes,' Dimes');
   WRITELN(Nickels, ' Nickels');
   WRITELN(Cents, ' Cents')
END.
```

A sample run of the program gives the following:

```
Enter the total-price in cents: 22
The change is:
3 Quarters
0 Dimes
0 Nickels
3 Cents
```

DRILL 2-1

Modify the last program to accept any amount of money as total-price (including fractions of a dollar) and any amount of cash as amount-paid.

The program should read the amount-paid and the total-price and display the change in bills of different denominations, quarters, dimes, nickels, and cents.

TURBO PASCAL ADDITIONAL FUNCTIONS Turbo Pascal has a considerable number of additional arithmetic functions. Of these functions, you will especially need two of them:

FRAC(n)	returns the fractional portion of the real number "n"
INT(n)	returns the integer portion of the real number "n"

For example:

`WRITELN(FRAC(8.22):2:2);`	produces 0.22
`WRITELN(INT(8.22)2:2);`	produces 8.00

Both functions return real numbers. You can make use of these functions in drill 2-1.

Another couple of functions are used to generate random numbers:

RANDOM(n)	returns a random integer between 0 and the integer "n" (the zero is included).
RANDOM	returns a real random number between 0 and 1 (the zero is included).

Try these two statements:

```
WRITELN(RANDOM:2:2);
WRITELN(RANDOM(n));
```

where "n" is an integer variable readout from the keyboard.

Use the two statements in a program and look at the results for several runs. They should be different in each run.

DRILL 2-2

Write the Pascal expressions for the following:

1. The quadratic equation: $Ax^2 + Bx + C$
2. The determinant: $B^2 - 4AC$
3. The square root of the determinant
4. The absolute value of the determinant

Then, write a program to produce the roots of the equation according to the input values of A, B, and C. Use test values for A, B, and C that give real roots. Typical values are:

A=1, B=2, C=1, give the solution: X1= X2= –1.00
A=1, B=4, C=2, give the solution: X1= –0.59, X2= –3.41

2-4 THE CHARACTER TYPE CHAR

The CHAR type is used to store a single character in Pascal. You can declare a variable of the type CHAR as in the following example:

```
VAR
    SingleLetter : CHAR;
```

In the main body of the program (between BEGIN and END.) you may assign a single character to the variable "SingleLetter" like this:

```
    SingleLetter:='A';
```

As is clear from this example, a constant literal of the type CHAR must be exactly one character, included in single quotes:

'A'   '3'   '*'   '$'   ' '

In order to represent a single quotation (or apostrophe) as a character constant, use two single quotes like this:

```
''''
```

You can use the output statements WRITELN or WRITE to display a character constant or a character variable:

```
WRITELN('A');
WRITELN(SingleLetter);
```

The character set is internally represented by a one-byte integer code. The universally used code for small computers is the ASCII code (American Standard Code for Information Interchange). The ASCII code includes 256 characters from 0 to 255 (see Appendix A). The first half of the ASCII code (from 0 to 127) is standard on all personal computers. It includes the following characters:

- The uppercase letters (A-Z): ASCII 65 to 90
- The lowercase letters (a-z): ASCII 97 to 122
- The digits (0-9): ASCII 48 to 57

The code also contains punctuation characters and control characters.

The second half of the ASCII code is not standard and is implemented differently on different machines.

The relative sequence of a character in the ASCII set is called the *ordinal number*.

STANDARD FUNCTIONS FOR CHARACTERS There are four standard functions that are dedicated to handle character operations:

ORD(c)	returns the ordinal number of the character "c."
CHR(n)	returns the character represented by the ordinal number "n."
PRED(c)	returns the character preceding "c" in the ordinal sequence.
SUCC(c)	returns the next character after "c" in the ordinal sequence.

You can get the ordinal number of any character by using the function ORD, as in the following example:

```
WRITELN(ORD('A'));
```

This statement displays the ordinal of the character "A," which is 65.

In the following program the user enters a character and the program displays the corresponding ordinal number.

```
{ ------------------------------ figure 2-4 ------------------------------ }
{ Displaying the Ordinal Number of a Character }
PROGRAM OrdinalNumber(INPUT,OUTPUT);
VAR
   SingleChar :CHAR;
BEGIN
   WRITE('Give me a character, please: ');
   READLN(SingleChar);
   WRITELN('The ordinal number of this character is ', ORD(SingleChar));
   READLN            { The program will pause until you press Enter }
END.
```

A sample run of the program gives the following:

```
Give me a character, please: A                ----> Type A and press Enter
The ordinal number of this character is 65 ----> The program output
```

TIP Notice the use of the last READLN statement. When READLN is used without parentheses, it holds the program until you press Enter. You cannot use READ for this purpose. This type of READLN statement is commonly preceded by a user prompt such as:

```
WRITELN('Press ENTER to continue');
```

The counterpart of ORD is the function CHR, which takes an ordinal number as a parameter and returns the character that corresponds to this number. Look at this example:

```
WRITELN(CHR(66));
```

This statement displays the letter "B."

In the following program, the user enters an ordinal number and the program displays the corresponding character.

```
{ ------------------------------ figure 2-5 ------------------------------ }
{ Displaying the Character, Knowing its Ordinal Number }
PROGRAM CharDisplay(INPUT,OUTPUT);
VAR
   OrdinalNum :BYTE;
BEGIN
   WRITE('Give me a number between 0 and 255: ');
   READLN(OrdinalNum);
   WRITELN('This corresponds to the character "', CHR(OrdinalNum),'"');
```

```
   WRITELN('Press ENTER to continue ...');
   READLN              { The program will pause until you press Enter }
END.
```

A sample run of the program gives the following:

```
Give me a number between 0 and 255: 66     ----> Enter the number 66
This corresponds to the character "B"      ----> The program output
Press ENTER to continue ...
```

NOTE Notice the use of the Turbo Pascal type BYTE to store an ordinal number, which is a positive integer between 0 and 255. If you don't have this type in your compiler, you can use the INTEGER type.

The following program demonstrates the use of the functions PRED and SUCC. You enter a character and the program displays the previous and the next characters.

```
{ ---------------------------- figure 2-6 ----------------------------- }
{ The Predecessor and the Successor to a Character }
PROGRAM CharPredAndSucc(INPUT,OUTPUT);
VAR
   Letter: CHAR;
BEGIN
   WRITE('Please Enter a character: ');
   READLN(Letter);
   WRITELN('The Predecessor to this character is "',PRED(Letter),'"');
   WRITELN('The Successor to this character is "',SUCC(Letter),'"');
   WRITELN('Press ENTER to continue ...');
   READLN
END.
```

A sample run gives the following:

```
Please Enter a character:K                 ----> Enter the character "K"
The Predecessor to this character is "J"   ----> The program response
The Successor to this character is "L"
Press ENTER to continue ...
```

You can use numbers or any special symbols from your keyboard to test this program. Remember, though, that some machines (mainframes) use a different sequence known as the EBCDIC code.

You may also use the function ORD with the type INTEGER, in which case it returns the sequence of the integer in the set of integers (from -(MAXINT+1) to MAXINT). Thus:

```
ORD(0)=0, ORD(1)=1, ORD(255)=255, and ORD(-22)=-22
```

The functions SUCC and PRED work with integers in the same way, which means:

```
SUCC(1)=2, and 'PRED(1)=0'
```

Some programmers increment their counters with a statement like this:

```
Counter := SUCC(Counter);
```

If you replace the type CHAR by the type INTEGER in the last program (figure 2-6), you can test these relations.

STRINGS IN STANDARD PASCAL As mentioned earlier, you can represent a string constant using single quotes like this:

```
'This is a string enclosed in single quotes'
```

To include an apostrophe in the string constant, you need two of them:

```
'This is an apostrophe '' included in a string'
```

You can also assign a string to a named constant:

```
CONST
   Name = 'Sally Shuttleworth';
```

After this declaration you can use the named constant "Name" instead of the string itself, but remember that in the program you cannot assign any value to a named constant. The string variable, however, is not defined in standard Pascal. A string, in standard Pascal, is stored in a PACKED ARRAY OF CHAR which is declared like this:

```
VAR
   Name : PACKED ARRAY[1..15] OF CHAR;
```

This declaration lets you store a string of exactly 15 characters in the variable "Name" — no more, no less.

Look at the following example, where the variable "Message" is declared and assigned the string 'Press any key ...   '. Extra spaces are padded to the end of the string constant to make it fit into the variable "Message," which was declared as a PACKED ARRAY OF CHAR 21 characters long.

```
{ ---------------------------- figure 2-7 ---------------------------- }
{ Packed Array Of Characters }
PROGRAM PackedArray(OUTPUT);
VAR
   Message :PACKED ARRAY[1..21] OF CHAR;
BEGIN
   Message := 'Press any key ...      ';
   WRITELN(Message)
END.
```

The output is:

```
Press any key ...
```

2-5 THE STRING TYPE

Actually, you will never need to use the PACKED ARRAY OF CHAR unless you are using one of the old implementations of Pascal on a mainframe computer. In the modern implementations (such as Turbo and UCSD), the type STRING is defined.

DECLARATION OF A STRING You can declare a variable of the type string, as in this example:

```
VAR
   StudentName : STRING;
```

This declaration lets you store a string of up to a certain size in the variable "StudentName." Although the maximum *length* of the string variable is 255 in Turbo (80 in UCSD), the actual length (also referred to as *dynamic length*) of the string is the number of stored characters. You can declare the string variable and its maximum length in the same statement:

```
VAR
   StudentName : STRING[20];
```

In this case the maximum length of a string stored in the variable "StudentName" is 20 characters. Look at this program, which reads a name of a maximum length of 20 characters and displays it on the screen.

```
{ ---------------------------- figure 2-8 ---------------------------- }
{ String Type in Turbo Pascal }
PROGRAM StringDemo(INPUT,OUTPUT);
VAR
```

```
   Name :STRING[20];
BEGIN
   WRITE('Please enter a name of 20 characters or less:');
   READLN(Name);
   WRITELN('The name you entered is ',Name, '. Is that right?')
END.
```

A sample run of the program gives the following:

```
Please enter a name of 20 characters or less:Peter Rigby
The name you entered is Peter Rigby. Is that right?
```

Note that if you assign a string constant of more than 20 characters to the variable "Name," the extra characters will be truncated.

THE LENGTH OF A STRING You can measure the dynamic length of a string using the function LENGTH. If you want, for instance, to measure the length of the string "Name" in the last program, you may use the expression:

```
LENGTH(Name)
```

If you display the value of this expression, you get the exact number of characters contained in the string variable, including the spaces. If the string variable is empty, the dynamic length is zero. In the following program, you enter a name and the program displays the actual length both before and after the variable assignment.

```
{ ----------------------------- figure 2-9 ----------------------------- }
{ Dynamic Length of a String }
PROGRAM StringLen(INPUT,OUTPUT);
VAR
   Name :STRING[20];
BEGIN
   WRITELN('The dynamic length of the string is now ',LENGTH(Name),
                      ' characters');
   WRITE('Please enter a name of 20 characters or less:');
   READLN(Name);
   WRITELN('The dynamic length of the string is now ',LENGTH(Name),
                      ' characters')
END.
```

The following is a sample run:

```
The dynamic length of the string is now 0 characters
Please enter a name of 20 characters or less:Dale Sanders
The dynamic length of the string is now 12 characters
```

The introduction of the type string in Pascal filled a gap and added a powerful tool, especially in the field of text processing. More on string functions and operations later.

2-6 THE BOOLEAN TYPE

The BOOLEAN values (sometimes called logical values) are the two constants:

TRUE and FALSE

They are named after the English mathematician George Boole (1815-1864).

In Pascal you can declare a variable of the type BOOLEAN, which may only hold one of the two BOOLEAN constants TRUE or FALSE, as in the following example:

```
VAR
  Result : BOOLEAN;
```

SIMPLE BOOLEAN EXPRESSIONS You can assign a BOOLEAN constant to a BOOLEAN variable, such as:

```
Result := TRUE;
```

You may also assign a boolean expression to a variable such as:

```
Result := A > B;
```

If A, for example, holds the value 22.5 and B holds the value 2.3, then the expression "A > B" (A is larger than B) is evaluated as TRUE. If A holds 1.8, then the condition is not satisfied and the expression is evaluated as FALSE. You can build boolean expressions using the *relational operators* shown in table 2-4.

Table 2-4 Relational Operators

Operator	*Meaning*	*Example*
>	Greater than	A > B
<	Less than	C < 54
>=	Greater than or equal	x >= 16.8
<=	Less than or equal	A+B <= 255
=	Equal	SQR(B) = 4*A*C
<>	Not equal	CHR(a) <> 'N'

Relational operators are used with any data type: numeric, character, or BOOLEAN. Here are some examples:

Numeric:	y > 66.5 Y = A * x + B	
Character:	FirstCharacter = 'B' CHR(x) > 'A'	
Boolean:	TRUE > FALSE TRUE < FALSE	(always TRUE) (always FALSE)

For characters, an expression such as:

```
'A' < 'B'
```

is always TRUE, because the letter "A" comes before "B" in the alphabet; in other words, it has a smaller ordinal number. Using the same logic, the following expressions are TRUE:

```
'9' > '1'
'Y' < 'Z'
```

The following program reads from the keyboard the value of two integers "A" and "B" and displays the value of the boolean expression "A = B."

```
{ ----------------------------- figure 2-10 ---------------------------- }
{ Boolean Variables }
PROGRAM Compare1(INPUT,OUTPUT);
VAR
   A, B   :INTEGER;
   Result :BOOLEAN;
BEGIN
   WRITE('Please enter two integers: ');
   READLN(A, B);
   Result := (A = B);
         { or,
           Result := A = B;
           The parentheses are not necessary. }
   WRITELN('The comparison is ', Result)
END.
```

The following are two sample runs of the program:

```
Please enter two integers: 5 5
The comparison is TRUE
Please enter two integers: 50 55
The comparison is FALSE
```

As mentioned earlier, you may not compare two real values for equality because of their limited precision. In the following program, the difference between the two real variables "x" and "y" is tested to see whether it is less than a specific small value "Difference," in which case they are considered to be equal.

```
{ ------------------------------ figure 2-11 ---------------------------- }
{ Comparing real values }
PROGRAM Compare2(INPUT,OUTPUT);
CONST
   Difference = 0.0001;
VAR
   x, y   :REAL;
   Result :BOOLEAN;
BEGIN
   WRITE('Please enter two real numbers: ');
   READLN(x, y);
   Result := ABS(x - y) < Difference;
   WRITELN('The difference is ', ABS(x-y):2:6);
   WRITELN('The comparison is ', Result)
END.
```

The following is a sample run:

```
Please enter two real numbers: 4.5 4.50001
The difference is 0.000010
The comparison is TRUE
```

COMPOUND BOOLEAN EXPRESSIONS The boolean expressions which use relational operators are called *simple boolean expressions* (in other languages they are called relational expressions). The *compound boolean expressions* are those which use the *boolean operators* (also called the logical operators): AND, OR, and NOT.

To understand how a compound boolean expression works, consider the example:

```
(x = 4) AND (y < 50)
```

This expression is evaluated TRUE if both conditions "x = 4" and "y < 50" are TRUE.

Now consider the same expression using the operator OR:

```
(x = 4) OR (y < 50)
```

This expression is evaluated as TRUE if any one of the conditions is TRUE. For example, if "x" contains the value "4," the expression is TRUE regardless of the value of "y."

The logical operator NOT is used to reverse the value of a boolean expression. Suppose that the boolean variable "UnderAge" means that the age is less than 18, as in the following statement:

```
UnderAge := Age < 18;
```

The variable "UnderAge" will contain the value TRUE if the "Age" is less than 18.

Now the expression:

```
NOT(UnderAge)
```

is evaluated TRUE if the "Age" is 18 or above.

TURBO PASCAL LOGICAL OPERATORS Turbo Pascal also contains the logical operator XOR, which is called the exclusive OR. It is used as in the following expression:

```
(x = 4) XOR (x = 400)
```

The value of this expression is TRUE if either one of the two conditions ("x = 4" or "x = 400") is TRUE, but the expression is evaluated as FALSE if both conditions are either TRUE or FALSE. In any implementation of Pascal you can use the operator <> as the *exclusive* OR. You can write the previous expression as:

```
(x = 4) <> (x = 400)
```

PRECEDENCE OF OPERATORS As with arithmetic expressions, the precedence of operators should be considered when building a boolean expression (relational or logical). Table 2-5 summarizes the relative precedence of all operators you have used so far.

Table 2-5 Precedence of Pascal Operators

Operator	*Precedence*
NOT	Priority 1 (highest)
* / DIV MOD AND	Priority 2
+ − OR (XOR in Turbo Pascal)	Priority 3
= > < >= <= <>	Priority 4 (lowest)

To understand the effects of precedence, try the boolean expression:

```
x = 4 OR x = 400
```

Because the OR has a higher precedence level than the equality, this will not compile because it will be interpreted as:

```
x = (4 OR x) = 400
```

which is not a valid expression.

DRILL 2-3

Write boolean expressions to express the following conditions:

1. A is less than 55.5
2. x is equal to y, or x is greater than or equal to z
3. either x=40, or y=80; or both
4. either x=40, or y=80; but not both

SUMMARY

1. In this chapter you learned the four standard data types:
   - **INTEGER**
   - **REAL**
   - **CHAR**
   - **BOOLEAN**
2. You also learned the additional numeric types of Turbo Pascal:

   I. Integers:
   - **SHORTINT**
   - **BYTE**
   - **INTEGER**
   - **WORD**
   - **LONGINT**

   II. Reals
   - **SINGLE**
   - **REAL**
   - **DOUBLE**

- **EXTENDED**
- **COMP**

3. You learned the standard arithmetic functions, classified into three groups:

   Conversion:

   - **ROUND**
   - **TRUNC**

   Trigonometric:

   - **ARCTAN**
   - **COS**
   - **SIN**

   Miscellaneous:

   - **ABS**
   - **EXP**
   - **LN**
   - **SQR**
   - **SQRT**

4. You also learned some additional arithmetic functions from Turbo Pascal, such as:

   - **FRAC**
   - **INT**
   - **RANDOM**

5. You can now write mathematical expressions using arithmetic operators and functions.

6. You are now familiar with four functions used to manipulate characters:

   - **CHR**
   - **ORD**
   - **PRED**
   - **SUCC**

7. You learned some of the features of text string variables in standard Pascal, and you know that such variables are defined as **PACKED ARRAYS OF CHAR**. In extensions such as Turbo Pascal and UCSD Pascal, the type **STRING** was added to the language along with other features and functions.

You learned the STRING function:

LENGTH

which is used to measure the dynamic length of a string.

8. Using the arithmetic, relational, and boolean operators, you learned how to build simple and compound boolean expressions and how to use the type **BOOLEAN**.

   You know as well the boolean operators:

   - **NOT**
   - **AND**
   - **OR**

   and you can express the exclusive OR in two ways:

   - using the relational operator <>
   - using the Turbo Pascal operator **XOR**

9. Finally, you had one last tour of Pascal operators and learned about their relative precedence.

CHAPTER THREE

DECISIONS

3-1 MAKING DECISIONS

So far, each of the programs in this book has been a series of instructions executed sequentially one after the other. In real-life applications, however, you will usually need to change the sequence of execution according to specified conditions. Sometimes you need to use a simple condition like:

> "**If** it is cold **then** put your coat on."

In this statement the resultant action is taken if the condition is evaluated as TRUE (the weather is cold). If, however, the weather was fine, the whole statement would be skipped.

Some conditions could be multiple, like those in the following conversation:

> "Ok then, **if** I come back early from work, I'll see you tonight;
> **else if** it is too late I'll make it tomorrow; **else if** my brother arrives tomorrow we can get together on Tuesday; **else if** Tuesday is a holiday then let it be Wednesday; **else** I'll call you to arrange for the next meeting!"

Actually, your program can easily handle such chained or nested conditions as long as you write the adequate code.

In Pascal there are two *control structures* used to handle conditions and their resultant decisions: the binary choice construct **if-then-else**, and the multiple choice construct **case**.

3-2 THE SIMPLE DECISION: IF-THEN

To express a simple condition you can use the **if-then** statement, as in the following example:

```
IF Age < 18 THEN
  WRITELN('Sorry, this is underage.');
```

The statement starts with the keyword IF, followed by a boolean expression (the condition to be tested), followed by the keyword THEN, followed by the result to be executed if the condition is TRUE (the WRITELN statement). As you can see, the IF construct is one statement ending with a semicolon. If the value of variable "Age" is less than 18, the part after the keyword THEN is executed; otherwise, the whole statement is skipped, and the program execution resumes its original flow at the next statement. This type of program control is called conditional branching.

The IF-THEN statement takes the general form:

IF condition **THEN**
  statement;

The construct is written in two lines just for readability, but it is one statement ending with a semicolon, and there is no obligation to leave extra spaces. You only need to separate the keywords (such as IF and THEN) from the rest of the statement by at least one space.

EXAMPLE: PASCAL CREDIT CARD Take a look at the following program, where a credit card limit is tested for a certain purchase. The program starts with declaration of the constant "Limit" which represents the credit card limit ($1000), and the variable "Amount" whose value will be received from the keyboard. The program displays the message "Your charge is accepted" if the "Amount" is less than or equal to the "Limit." If the condition is FALSE the program will end without response.

```
{ ------------------------------ figure 3-1 ------------------------------ }
PROGRAM SimpleDecision(INPUT,OUTPUT);
CONST
  Limit = 1000;
VAR
  Amount :REAL;
BEGIN
  WRITE('Please enter the amount:');
  READLN(Amount);
```

```
   IF Amount <= Limit THEN
      WRITELN('Your charge is accepted.');{End of the IF statement}
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

A READLN statement is used to pause the screen while displaying the message "Press ENTER to continue." Because this statement is outside the IF statement it will be executed whether the condition is TRUE or FALSE. Here are two sample runs:

RUN 1:

```
Please enter the amount:200
Your charge is accepted.
Press ENTER to continue..
```

RUN 2:

```
Please enter the amount:2000
Press ENTER to continue..
```

You can use two conditional statements to represent the two cases, the TRUE and the FALSE. In the following program another IF statement is added to deal with the other case (the amount is greater than 1000). The message "The amount exceeds your credit limit" is displayed in this case.

```
{ ----------------------------- figure 3-2 ----------------------------- }
PROGRAM TwoConditions(INPUT,OUTPUT);
CONST
   Limit = 1000;
VAR
   Amount :REAL;
BEGIN
   WRITE('Please enter the amount:');
   READLN(Amount);
   IF Amount <= Limit THEN
      WRITELN('Your charge is accepted.');
   IF Amount > Limit THEN
      WRITELN('The amount exceeds your credit limit.');
   WRITELN('Thank you for using Pascal credit card.');
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following are two sample runs of the program:

RUN 1:

```
Please enter the amount:15C
Your charge is accepted.
Thank you for using Pascal credit card.
Press ENTER to continue..
```

RUN 2:

```
Please enter the amount:1500
The amount exceeds your credit limit.
Thank you for using Pascal credit card.
Press ENTER to continue..
```

As before, note that the last two lines were displayed in each case, as they do not belong to the conditional statements.

USING BLOCKS If you want to use more than one statement as a result of one condition, you can use the BEGIN-END blocks. You can actually use any number of blocks inside the program main body, using BEGIN and END to mark the territories of each block. A block will be treated as one unit, no matter how many statements it includes. Look at the following example:

```
{ ------------------------------ figure 3-3 ------------------------------ }
PROGRAM UsingBlocks(INPUT,OUTPUT);
CONST
   Limit = 1000;
VAR
   Amount :REAL;
BEGIN
   WRITE('Please enter the amount:');
   READLN(Amount);
   IF Amount <= Limit THEN
      BEGIN
         WRITELN('Your charge is accepted.');
         WRITELN('Your price plus tax is $',1.05*Amount:0:2)
                         { The semicolon is optional }
      END;
   IF Amount > Limit THEN
      BEGIN
         WRITELN('The amount exceeds your credit limit.');
         WRITELN('The maximum limit is $',Limit)
                         { The semicolon is optional }
      END;
   WRITELN('Thank you for using Pascal credit card.');
```

```
   WRITELN('Press ENTER to continue..');
   READLN               { The semicolon is optional }
END.
```

In this example more than one statement is executed in either case (TRUE or FALSE). For this reason two blocks were used.

TIP Notice that in three positions in this program, the statement is not terminated by a semicolon, as the semicolon is optional. The statement in each of these positions is the last one inside a block.

Here are two sample runs:

RUN 1:

```
Please enter the amount:120
Your charge is accepted.
Your price plus tax is $126.00
Thank you for using Pascal credit card.
Press ENTER to continue..
```

RUN 2:

```
Please enter the amount:2000
The amount exceeds your credit limit.
The maximum limit is $1000
Thank you for using Pascal credit card.
Press ENTER to continue..
```

If you try the program without the blocks, you will find that only the first statement that follows the keyword THEN belongs to the IF statement, but any other statement belongs to the main program and will be executed regardless of the condition.

DRILL 3-1

Write a program to accept from the keyboard a character and test this character to see if it is one of the following:

1. A number
2. A lowercase letter
3. An uppercase letter

Display the suitable message in each case.

3-3 THE IF-ELSE-THEN CONSTRUCT

The form you have used so far for the IF statement is actually a simplified version of the complete construct. The complete IF statement includes the two cases that result from testing the condition. It takes the form:

```
IF condition THEN
   statement
ELSE
   statement;
```

Notice here that only one semicolon is used, because the whole construct is treated as one statement. Here is an example:

```
IF AGE < 18 THEN
   WRITELN('Underage.')
ELSE
   WRITELN('Age is OK.');
```

This statement will display the message "Underage" if "Age" is less than 18. In the other case the message "Age is OK" is displayed.

If you add another statement to either of the two cases, you have to use the BEGIN-END blocks. The new construct will look like this:

```
IF AGE < 18 THEN
   BEGIN
      WRITELN('Underage.');
      WRITELN('Wait another couple of years.')
   END                   { No semicolon is used here }
ELSE
   BEGIN
      WRITELN('Age is OK.');
      WRITELN('You don''t have to wait.')
END;                     { A semicolon is mandatory here }
```

CAUTION

At this point the use of semicolons becomes critical and may lead to errors if not done properly. Notice here that the keyword END in the first block is not terminated by a semicolon (as it is not the end of the statement), while in the second block it is terminated by a semicolon, indicating the end of the conditional statement.

Now, back to the "Pascal credit card" program to enhance it with the complete IF-THEN-ELSE statement.

```
{ ----------------------------- figure 3-4 ----------------------------- }
PROGRAM CreditCard(INPUT,OUTPUT);
CONST
   Limit = 1000;
VAR
   Amount :REAL;
BEGIN
   WRITE('Please enter the amount:');
   READLN(Amount);
{ Beginning of the IF construct }
{ ---------------------------- }
   IF Amount <= Limit THEN
      BEGIN
         WRITELN('Your charge is accepted.');
         WRITELN('Your price plus tax is $',1.05*Amount:0:2)
      END
   ELSE
      BEGIN
         WRITELN('The amount exceeds your credit limit.');
         WRITELN('The maximum limit is $',Limit)
      END;
{ End of the IF construct }
{ ----------------------- }
   WRITELN('Thank you for using Pascal credit card.');
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

Sample runs of the program give the following results:

RUN 1:

```
Please enter the amount:1000
Your charge is accepted.
Your price plus tax is $1050.00
Thank you for using Pascal credit card.
Press ENTER to continue..
```

RUN 2:

```
Please enter the amount:1001
The amount exceeds your credit limit.
The maximum limit is $1000
```

```
Thank you for using Pascal credit card.
Press ENTER to continue..
```

DRILL 3-2

Modify the program you wrote in drill 2-2 to solve a quadratic equation ($Ax^2 + Bx + C$) for both real and imaginary roots.

3-4 THE ELSE-IF LADDER

Although the IF-THEN-ELSE statement is intended for binary choice, it can be extended to handle more complicated choices. Look at this new arrangement of the construct, which is sometimes referred to as the ELSE-IF ladder:

```
IF condition-1 THEN
   statement-1
ELSE IF condition-2
   statement-2
ELSE IF condition-3
   statement-3
...
ELSE
   statement-n;
```

The conditions in the ladder are evaluated from the top down, and whenever a condition is evaluated as TRUE, the corresponding statement is executed and the rest of the construct is skipped. If no condition has been satisfied, the last ELSE will be brought into action.

Notice that the condition ladder is considered one statement ending with a semicolon, but no semicolons are used inside. If you want to use more than one result-statement, you have to use the BEGIN-END blocks according to the rules mentioned earlier.

EXAMPLE: A CHARACTER TESTER This program starts by asking you to enter a letter, then tests the input character to see if it is a lowercase or uppercase letter. The program can also recognize numbers and deliver an appropriate message, but otherwise it displays: "Sorry, this is not a letter."

The logic used in the program depends on testing the ASCII code of the input characters using the ORD function. The characters are classified as follows:

A. The uppercase letters correspond to the codes from 65 to 90.

B. The lowercase letters correspond to the codes from 97 to 122.

C. The digits correspond to the codes from 48 to 57.

If you already wrote this program as a solution to drill 3-1, you will find that the ELSE-If ladder makes things easier.

```
{ ------------------------------ figure 3-5 ------------------------------
PROGRAM CharsTester(INPUT,OUTPUT);
VAR
   InputChar :CHAR;
BEGIN
   WRITE('Please enter an alphabetic character:');
   READLN(InputChar);
{ Beginning of the IF construct }
{ ----------------------------- }
   IF (ORD(InputChar) > 64) AND (ORD(InputChar) < 91) THEN
      WRITELN('This is an upper-case letter.')
   ELSE IF (ORD(InputChar) > 96) AND (ORD(InputChar) < 123) THEN
      WRITELN('This is a lower-case letter.')
   ELSE IF (ORD(InputChar) > 47) AND (ORD(InputChar) < 58) THEN
      WRITELN('Hey, this is a number!')
   ELSE
      WRITELN('Sorry, this is not a letter.');
{ End of the IF construct }
{ ----------------------- }
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following are four sample runs for four different inputs:

RUN 1:

```
Please enter an alphabetic character:a       ----> Enter "a"
This is a lower-case letter.
Press ENTER to continue..
```

RUN 2:

```
Please enter an alphabetic character:B       ----> Enter "B"
This is an upper-case letter.
Press ENTER to continue..
```

RUN 3:

```
Please enter an alphabetic character:5        ----> Enter "5"
Hey, this is a number!
Press ENTER to continue..
```

RUN 4:

```
Please enter an alphabetic character:@        ----> Enter "@"
Sorry, this is not a letter.
Press ENTER to continue..
```

3-5 NESTED CONDITIONS

The statement to be executed upon testing a condition can be of any kind. As a matter of fact, it can be another IF statement nested in the original IF statement.

The IF-THEN-ELSE constructs can be nested inside each other, as in the following form:

```
IF condition-1 THEN
   IF condition-2 THEN
      ...
      IF condition-n THEN
        statement-n1
      ELSE
        statement-n2
      ...
   ELSE
     statement-2
ELSE
  statement-1;
```

As you can see, this construct can handle any number of nested conditions, but you have to keep track of each IF and the corresponding ELSE. Let us put the construct into action.

EXAMPLE: SCORES AND GRADES This program receives the score of a student and displays the grade according to the following classification:

1. Grade "A" corresponds to scores from 90% to 100%.
2. Grade "B" corresponds to scores from 80% to 89%.

3. Grade "C" corresponds to scores from 70% to 79%.
4. Grade "D" corresponds to scores from 60% to 69%.
5. Grade "F" corresponds to scores less than 60%.

Here is the program:

```
{ ----------------------------- figure 3-6 ----------------------------- }
PROGRAM ScoresAndGrades1(INPUT,OUTPUT);
VAR
   Score :INTEGER;
BEGIN
   WRITE('Please enter the score:');
   READLN(Score);
   WRITELN;
{ Beginning of the IF construct }
{ ---------------------------- }
   IF Score > 59  THEN
      IF Score > 69 THEN
         IF Score > 79 THEN
            IF Score > 89 THEN
               WRITELN('Excellent. Your grade is ''A''')
            ELSE
               WRITELN('Very good. Your grade is ''B''')
         ELSE
            WRITELN('Good. Your grade is ''C''')
      ELSE
         WRITELN('Passed. Your grade is ''D''')
   ELSE
      WRITELN('Better luck next time. Your grade is ''F''');
{ End of the IF construct }
{ ---------------------- }
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following are sample runs of the program:

RUN 1:

```
Please enter the score:92          ----> Enter "92"
Excellent. Your grade is 'A'       ----> The program response
Press ENTER to continue..
```

RUN 2:

```
Please enter the score:70
Good. Your grade is 'C'
Press ENTER to continue..
```

RUN 3:

```
Please enter the score:60
Passed. Your grade is 'D'
Press ENTER to continue..
```

RUN 4:

```
Please enter the score:59
Better luck next time. Your grade is 'F'
Press ENTER to continue..
```

As usual, you may cause more than one result statement to be executed upon testing a condition by embedding the statements into a block.

You can use any one of the available variations of the IF-THEN-ELSE construct in your applications. However, some forms are more reliable with one application, and some with others. Look at this program, which processes the same problem of the "scores and grades" but uses the ELSE-IF ladder. Notice how the program is made easier and more comprehensible to the reader by using the boolean variables "A," "B," "C," "D," "F." Note also that illegal numbers are filtered out by the last ELSE.

```
{ ------------------------------ figure 3-7 ----------------------------- }
PROGRAM ScoresAndGrades2(INPUT,OUTPUT);
VAR
   Score         :INTEGER;
   A, B, C, D, F :BOOLEAN;
BEGIN
   WRITE('Please enter the score:');
   READLN(Score);
   A := (Score >= 90) AND (Score <= 100);
   B := (Score >= 80) AND (Score < 90);
   C := (Score >= 70) AND (Score < 80);
   D := (Score >= 60) AND (Score < 70);
   F := (Score < 60)  AND (Score >= 0);
   WRITELN;
{ Beginning of the IF construct }
{ ----------------------------- }
   IF A THEN
```

```
      WRITELN('Excellent. Your grade is ''A''')
   ELSE IF B THEN
      WRITELN('Very good. Your grade is ''B''')
   ELSE IF C THEN
      WRITELN('Good. Your grade is ''C''')
   ELSE IF D THEN
      WRITELN('Passed. Your grade is ''D''')
   ELSE IF F THEN
      WRITELN('Better luck next time. Your grade is ''F''')
   ELSE
      WRITELN('This number is out of range.');
{ End of the IF construct }
{ ---------------------- }
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

TIPS ON THE IF-ELSE PUZZLES Nesting the IF constructs inside each other may become confusing (to the programmer), as one may not be able to tell which ELSE belongs to which IF. Look at this simple example:

```
IF X >= 1 THEN
IF y >= 18 THEN
WRITELN('statement#1.')
ELSE
WRITELN('statement#2');
```

The rule is that each ELSE belongs to the last IF in the same block. This means that, in this example, the ELSE belongs to the second IF. Arranging the text with the proper indentation, according to this rule, makes it clearer:

```
IF X >= 1 THEN
   IF y >= 18 THEN
      WRITELN('statement#1.')
   ELSE
      WRITELN('statement#2');
```

If, however, you want to associate ELSE with the first IF, you can use blocks as follows:

```
IF X >= 1 THEN
   BEGIN
      IF Y >= 18 THEN
         WRITELN('statement#1.')
   END
```

```
ELSE
   WRITELN('statement#2');
```

DRILL 3-3

Write a program to describe the weather according to the following temperature classifications:

Temperature	*Classification*
greater than 75	hot
50 to 75	cool
35 to 49	cold
less than 35	freezing

3-6 THE MULTIPLE CHOICE CASE

The CASE construct is used to deal with multiple alternatives, such as the user-menu options. It takes the general form:

CASE expression **OF**
  label-1 : statement-1;
  label-2 : statement-2;
  ...
  label-n : statement-n;
END

The *case expression* (also called the *selector*) can be of any type except REAL. According to the value of this expression the control of the program is transferred to one of the *case labels*, and the corresponding statement is executed. The labels actually represent the different possible values of the expression. Look at this example:

EXAMPLE: A VENDING MACHINE The coins in the vending machine are sorted according to the weight of each coin, which is assumed to be 35 grams for a quarter, 7 for a dime, and 15 for a nickel.

This logic can be programmed as follows:

```
CASE CoinWeight OF
   35 : Amount := Quarter;
   7  : Amount := Dime;
   15 : Amount := Nickel;
END;
```

The numbers 35, 7, and 15 represent the "CoinWeight" and are used as labels. Therefore, when the "CoinWeight" equals 7, for example, the statement:

```
Amount := Dime;
```

is executed. Needless to say, the name "Dime" is a named constant whose value is 10, and "Nickel" and "Quarter" are named constants as well. Look at the complete program:

```
{ ---------------------------- figure 3-8 ------------------------------ }
PROGRAM CaseOfWeights(INPUT,OUTPUT);
CONST
  Quarter = 25;
  Dime = 10;
  Nickel = 5;
VAR
  CoinWeight, Amount :INTEGER;
BEGIN
  WRITE('Please enter the weight:');
  READLN(CoinWeight);
  CASE CoinWeight OF
    35 : Amount := Quarter;
    7  : Amount := Dime;
    15 : Amount := Nickel;
  END;
  WRITELN('The amount is ', Amount, ' cents.');
  READLN
END.
```

This is a sample run of the program:

```
Please enter the weight:35      ----> Enter 35
The amount is 25 cents.         ----> The program response
```

You can use more than one label for the same result statement, which will save a lot of writing as compared to the IF in the same situation. Here is an example:

EXAMPLE: NUMBER OF DAYS IN A MONTH Consider, for instance, that you want to program a code that reads the number of the month and tells the number of days in that month. The CASE construct will look something like the following:

```
CASE Month OF
  1,3,5,7,8,10,12 : Days := 31;
  4,6,9,11        : Days := 30;
```

```
    2                 : Days := 28;
  END;
```

As you can see, the CASE construct here contains three cases, two of them with more than one label. All months that have 31 days belong to the first case, those that have 30 days belong to the second case, and February is a special case by itself. We assume here that February has 28 days for simplicity, but you can extend the logic to determine if the year is a leap year and assign February a value of 29 or 28 accordingly. You may use a block of statements for one case like this:

```
  CASE Month OF
  1,3,5,7,8,10,12 : Days := 31;
  4,6,9,11        : Days := 30;
  2               : BEGIN
                      WRITE('Enter the year:');
                      READLN(Year);
                      IF YEAR MOD 4 = 0  THEN
                        Days :=29
                      ELSE
                        Days :=28
                    END;
```

Here the case label "2" leads to a block of statements. So, if you enter "2" as the number of the month, the program will ask you to enter the year. The year will be tested and you will get 29 if the year is a leap year and 28 otherwise. Here is the complete program:

```
{ ----------------------------- figure 3-9 ----------------------------- }
PROGRAM DaysOfMonth1(INPUT,OUTPUT);
VAR
  Days, Month, Year :INTEGER;
BEGIN
  WRITE('Please enter the number of the month:');
  READLN(Month);
  CASE Month OF
    1,3,5,7,8,10,12 : Days := 31;
    4,6,9,11        : Days := 30;
    2               : BEGIN
                        WRITE('Enter the year:');
                        READLN(Year);
                        IF YEAR MOD 4 = 0  THEN
                          Days :=29
                        ELSE
                          Days :=28
```

```
                    END;
   END;
      WRITELN('There are ',Days,' days in this month.');
   READLN
END.
```

The following are sample runs of the program:

RUN 1:

```
Please enter the number of the month:2
Enter the year:1987
There are 28 days in this month.
```

RUN 2:

```
Please enter the number of the month:2
Enter the year:1984
There are 29 days in this month.
```

RUN 3:

```
Please enter the number of the month:12
There are 31 days in this month.
```

NOTE Notice that the leap year test in this program is a simplified logic, useful for the years of one century. The complete logic of the leap year definition is:

- The year is divisible by 4 AND not divisible by 100
  OR,
- The year is divisible by 400.

You may expand the logic as a drill.

In cases like this, using the CASE construct is more efficient than using nested IF-THEN-ELSE constructs or ladders. However, you must have realized that you will sometimes need them both (as in the February case).

DRILL 3-4

Write a program that reads the date from the keyboard in the form "dd mm yy" and displays the date as in the following examples:

January 2nd, 1992
October 23rd, 1990
March 5th, 1985

3-7 UNCONDITIONAL BRANCHING GOTO

The goto statement is used to transfer control of the program from one point to another. It is classified as *unconditional branching*. Although the goto statement is very easy to use, you rarely see it in Pascal programs because it destroys the structure of the program. In some cases, however, it may be useful in escaping from many levels of nesting in one jump. The syntax of the goto statement is as follows:

goto label;

The "label" is a positive integer of up to 4 digits preceding the required statement (in Turbo Pascal the label can be any valid identifier and may begin with a digit).

```
    GOTO 1000;
    ...
 1000:
    WRITELN('I am a labeled statement.');
    ...
```

When the goto is encountered, the program control is transferred to the labeled statement. The label must be declared in the *label section* of the declaration part of the program. The label section starts with the keyword LABEL and comes as the first section in the declaration part in standard Pascal (in Turbo Pascal there is no such obligation). Look at this example:

```
PROGRAM GoToDemo(INPUT,OUTPUT);
LABEL
   1000;
VAR
   InputChar :CHAR;
BEGIN
   WRITE('Please enter a letter (or 0 to quit):');
   READLN(InputChar);
   IF InputChar = '0'  THEN
      GOTO 1000;
   { Other statements
     may go here... }
1000:
   END.
```

In this example, the value of the input character is tested to see if it is zero, in which case control is transferred to the part following the label "1000," which

is the end of the program. If you are using Turbo Pascal, you can use meaningful labels such as "Wrapup" or "Start" instead of the numbers.

REPETITION LOOPS You can use the goto statement to build a closed loop. For example, if you want to repeat the execution of the "character tester" program, you may use the following logic, where the control is always transferred to the label "1000" at the beginning of the program. A condition is used to end the loop (and the program) by examining the input value. If a zero is entered, the control is transferred to the label "2000," ending the program. If you remove this condition from the program, it will be repeated infinitely. The only way to exit the program in this case is to use the control keys Ctrl-Break. This kind of loop is called an infinite loop.

```
{ ------------------------------ figure 3-10 ---------------------------- }
PROGRAM CharsTester2(INPUT,OUTPUT);
LABEL
   1000, 2000;                 { label declaration }
VAR
   InputChar :CHAR;
BEGIN
1000:
   WRITE('Please enter a letter (or 0 to quit): ');
   READLN(InputChar);
{ Beginning of the IF construct }
{ ---------------------------- }
   IF InputChar = '0'  THEN              { a condition to exit }
      GOTO 2000
   ELSE IF (ORD(InputChar) > 64) AND (ORD(InputChar) < 91) THEN
      WRITELN('This is an upper-case letter.')
   ELSE IF (ORD(InputChar) > 96) AND (ORD(InputChar) < 123) THEN
      WRITELN('This is a lower-case letter.')
   ELSE IF (ORD(InputChar) > 47) AND (ORD(InputChar) < 58) THEN
      WRITELN('Hey, this is a number!')
   ELSE
      WRITELN('Sorry, this is not a letter.');
{ End of the IF construct }
{ ----------------------- }
   GOTO 1000;             { restart the program }
2000:                     { exit the program }
END.
```

The following is a sample run of the program:

```
Please enter a letter (or 0 to quit):W      ----> Enter W
This is an upper-case letter.
Please enter a letter (or 0 to quit):e      ----> Enter e
This is a lower-case letter.
Please enter a letter (or 0 to quit):0      ----> Enter 0
```

This method, as you can see, is not the best method with which to build loops or control program execution, as it consists of jumps from one point to another. In the next chapter you are introduced to Pascal structured loops.

3-8 TURBO PASCAL FEATURES: EXIT, CASE-ELSE

If you entered an illegal value in program 3-9, such as the number 13 (as the month number), you simply get the message:

```
There are 0 days in this month.
```

In order to handle the invalid data you have to use a suitable IF statement. In Turbo Pascal you can add an ELSE part to the control structure CASE in order to handle data that do not belong to any of the case labels. The CASE structure will then take the form:

```
CASE expression OF
   label-1 : statement-1;
   label-2 : statement-2;
   ...
   label-n : statement-n;
ELSE
   statement
END
```

Another feature of Turbo Pascal is the EXIT statement, which ends the execution of the program at any point. The EXIT statement is classified as an unconditional branching statement. In the following program these two features are illustrated. If you enter any number rather than the numbers from 1 to 12, the ELSE part and the EXIT statement will end the program.

```
{ ----------------------------- figure 3-11 ---------------------------- }
PROGRAM DaysOfMonth2(INPUT,OUTPUT);
LABEL
  Start;
VAR
```

```
   Days, Month, Year :INTEGER;
BEGIN
Start:
   WRITE('Please enter the number of the month: ');
   READLN(Month);
   CASE Month OF
      1,3,5,7,8,10,12 : Days := 31;
      4,6,9,11        : Days := 30;
      2               : BEGIN
                          WRITE('Enter the year:');
                          READLN(Year);
                          IF YEAR MOD 4 = 0  THEN
                             Days :=29
                          ELSE
                             Days :=28
                        END;
   ELSE
      EXIT              { all other cases }
   END;
      WRITELN('There are ',Days,' days in this month.');
   GOTO Start
END.
```

This is a sample run:

```
Please enter the number of the month:1
There are 31 days in this month.
Please enter the number of the month:4
There are 30 days in this month.
Please enter the number of the month:13     ----> Exit the program
```

SUMMARY

In this chapter you learned the branching control structures that help you to handle decisions in your program.

1. You are now familiar with the simple IF-THEN statement used with simple decisions. It takes the form:

   IF condition **THEN**
      statement;

2. You also know the complete IF-THEN-ELSE construct that contains the result and the alternative result:

```
IF condition THEN
   statement
ELSE
   statement;
```

3. You also know how to handle complicated conditions using the ELSE-IF ladder in the form:

```
IF condition-1 THEN
   statement-1
ELSE IF condition-2
   statement-2
ELSE IF condition-3
   statement-3
...
ELSE
   statement-n;
```

4. An alternative to the ladder is nesting the IF-THEN-ELSE constructs inside each other in the form:

```
IF condition-1 THEN
   IF condition-2 THEN
      ...
      IF condition-n THEN
        statement-n1
      ELSE
        statement-n2
      ...
   ELSE
     statement-2
ELSE
  statement-1;
```

5. You learned how to use the multiple choice construct CASE, which is ready to handle many cases in the form:

```
CASE expression OF
   label-1 : statement-1;
   label-2 : statement-2;
   ...
   label-n : statement-n;
END
```

6. In Turbo Pascal the CASE construct has more features, as it may contain the ELSE part which handles all the other cases that do not correspond to a label. It takes the form:

   CASE expression **OF**
   label-1 : statement-1;
   label-2 : statement-2;
   ...
   label-n : statement-n;
   ELSE
   statement
   END

   You also understand that in any of the above formulas you can replace one statement by a block of statements using the BEGIN-END blocks.

7. You were introduced as well to the unconditional branching statement GOTO which transfers the program control to a labeled statement. It takes the form:

   GOTO label;

   The label in standard Pascal is a positive integer of up to 4 digits, while in Turbo Pascal it can be a valid identifier, or it may even begin with a number. You also know how to declare a label at the beginning of the declaration part of the program. In Turbo Pascal the LABEL section does not need to be the first section.

8. Finally, you met the Turbo Pascal statement EXIT, which terminates the program at any point.

In the next chapter, you continue the control structures to learn how to build structured loops.

DAY TWO

CHAPTER FOUR

LOOPS

4-1 LOOPING

You learned in the previous chapter how to build a repetition loop using the following tools:

- branching statement such as GOTO to transfer the control of the program to the starting point repeatedly
- condition to terminate the loop as desired

The condition may be used to test the input value and to terminate the loop when a specific value is received. You may also wish to repeat the process in the loop a specific number of times, in which case you need a counter. The condition in this case is used to test the counter with each round of the loop. This type of loop is called a *counted loop*. In the following program these elementary tools are used to display the message "Sorry, say again.." five times.

The algorithm used in the program is as follows:

A. Initialize the counter to zero.

B. Increment the counter by 1.

C. Test the counter to see if it is less than or equal to 5.

D. Display the statement.

E. Go to step B.

```
{ ----------------------------- figure 4-1 ----------------------------- }
PROGRAM GoToLoop(OUTPUT);
LABEL
   1000;              { label declaration }
VAR
```

```
  Kounter :INTEGER;
BEGIN
  Kounter := 0;
1000:
  Kounter := Kounter + 1;
  IF Kounter <= 5 THEN
    BEGIN
      WRITELN('Sorry, say again..');
    GOTO 1000         { restart }
    END;
  WRITELN;
  WRITELN('Press ENTER to continue..');
  READLN
END.
```

In this program the counter is initialized to the value zero before entering the loop, which begins at the label 1000. Inside the loop, the counter is incremented, then tested to see if its value is less than or equal to 5. If so, the WRITELN statement is executed and the loop is repeated using the GOTO statement. If the condition fails (i.e. the counter exceeds 5), the program ends. The output of this program looks like this:

```
Sorry, say again..
Sorry, say again..
Sorry, say again..
Sorry, say again..
Sorry, say again..
Press ENTER to continue..
```

Pascal provides you with ready-made control structures for looping, so you can avoid such messy code. A control structure contains both the branching statement and the condition in one construct.

In this chapter you are introduced to the following constructs:

- The FOR loop
- The WHILE loop
- The REPEAT loop

Each of the three loops has different features that suit different applications.

4-2 THE FOR LOOP

The FOR loop construct is a counted loop used to repeat a statement or a block of statements a specified number of times. It includes the initialization of the counter, the condition, and the increment.

Look at this example:

```
{ ------------------------------ figure 4-2 ------------------------------ }
PROGRAM ForLoop(OUTPUT);
VAR
   Kounter :INTEGER;
BEGIN
   FOR Kounter := 1 TO 5 DO
      WRITELN('Sorry, say again..');
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

This program gives the same results as the previous program does, but is simpler and better organized. The FOR loop does the same work done in the previous program. It assigns the *control variable* (Kounter) the *initial value* "1," then executes the statement, increments the control variable by one, and repeats the process until the value of the "Kounter" reaches the *final value* "5."

The general form of the FOR construct is as follows:

> **FOR** control-variable := expression-1 **TO** expression-2 **DO**
>   statement;

where:

control-variable	is the loop counter,
expression-1	is the initial value, and
expression-2	is the final value.

The control-variable, expression-1, and expression-2 can be of any type except REAL. All three must be of the same type.

TIP Remember that the FOR construct is one statement ending with a semicolon. If by mistake you add another semicolon, as in the following loop:

```
FOR Kounter := 1 TO 1000 DO;
     WRITELN('Sorry, say again..');
```

do not be surprised if the loop is executed only once, regardless of the final value of the counter. The semicolon after the DO keyword ends the loop at this point.

The value of the control variable may not be modified inside the loop. Look at this assignment statement inside the loop:

```
FOR K := 1 TO 10 DO
    K := 2
...
```

Even if the compiler accepts this statement, it will repeal the effect of the loop counter as it sets it to the value "2" all the time. The same rule applies for the initial value and the final value of the control variable.

As usual, you can include as many statements as you want inside the loop by using the BEGIN-END blocks.

EXAMPLE: POWERS OF TWO The number "2" and its powers are very important numbers in the computer field. Some of the numbers, such as 1024 Bytes (equivalent to 1 KB) and 65,536 Bytes (64 KB) are commonly used. In the following program a FOR loop is used to display the powers of two, using the same logic which was used to calculate the power in program 2-2. The program output gives the "power" and the number "2" raised to this power. The initial and final values of the counter are supplied by the user during the execution. Thus, you can determine the range of numbers you would like to examine.

```
{ ----------------------------- figure 4-3 ----------------------------- }
PROGRAM ForLoop(INPUT, OUTPUT);
VAR
   Base, Power, Start, Final :INTEGER;
BEGIN
   Base := 2;
   WRITE('Enter starting exponent:');
   READLN(Start);
   WRITE('Enter ending exponent:');
   READLN(Final);
   WRITELN;
   WRITELN('Number      Power of two');
   FOR Power := Start TO Final DO
      BEGIN
         WRITE(Power:3);
         WRITELN(EXP(LN(Base)*Power):20:0)
```

```
      END;
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following is a sample run using exponent values from 1 to 20:

```
Enter starting exponent:1
Enter ending exponent:20

Number        Power of two
  1                  2
  2                  4
  3                  8
  4                 16
  5                 32
  6                 64
  7                128
  8                256
  9                512
 10               1024
 11               2048
 12               4096
 13               8192
 14              16384
 15              32768
 16              65536
 17             131072
 18             262144
 19             524288
 20            1048576

Press ENTER to continue..
```

DRILL 4-1

Write a program to test the leap years in the range from 1990 to 2000. Display on the screen each year and the test result as in the following output:

```
The year 1990 is not a leap year.
The year 1991 is not a leap year.
The year 1992 is a leap year.
The year 1993 is not a leap year.
The year 1994 is not a leap year.
```

```
The year 1995 is not a leap year.
The year 1996 is a leap year.
The year 1997 is not a leap year.
The year 1998 is not a leap year.
The year 1999 is not a leap year.
The year 2000 is a leap year.
```

EXAMPLE: THE AVERAGE The following program demonstrates data entry using a loop. It receives from the keyboard a series of numbers and calculates the sum and the average of the numbers. At the beginning of the program you are asked to enter the number of the elements "N," which is used as the final value of the counter. Inside the loop the sum is accumulated in the variable "Sum" using the statement:

```
Sum := Sum + Number;
```

When the loop exits, the average is calculated from the sum and the number of elements, using the statement:

```
Average := Sum / N;
```

Here is the program.

```
{ ------------------------------ figure 4-4 ------------------------------ }
PROGRAM AverageProg1(INPUT,OUTPUT);
VAR
  Average, Sum, Number :REAL;
  N, Kounter           :INTEGER;
BEGIN
  Sum := 0;
  WRITE('Enter Number of Elements:');
  READLN(N);
  FOR kounter := 1 TO N DO
    BEGIN
      WRITE('Enter Element #',kounter,': ');
      READLN(Number);
      Sum := Sum + Number        { The semicolon is optional }
    END;
  Average := Sum / N;
  WRITELN;
  WRITELN('Sum of Numbers = ', Sum:0:2);
  WRITELN('Average of Numbers = ', Average:0:2);
  WRITELN;
```

```
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

A sample run of the program gives the following:

```
Enter Number of Elements: 5
Enter Element #1: 1
Enter Element #2: 2
Enter Element #3: 3
Enter Element #4: 4
Enter Element #5: 5

Sum of Numbers = 15.00
Average of Numbers = 3.00

Press ENTER to continue..
```

Notice how the element numbers were displayed inside the loop using the values of the control variable "Kounter."

4-3 STEPPING UP AND STEPPING DOWN

In the previous examples, the FOR loop counter was always incremented. This means that the final value of the counter must be greater than the initial value, or else the loop will never be executed.

You can decrement the counter using an alternative form of the FOR loop, by replacing the keyword TO with the keyword DOWNTO as in the following form:

```
FOR control-variable := expression-1
         DOWNTO expression-2 DO
   statement;
```

With this formula you can start the counter with the larger value and step down until the final value is reached.

EXAMPLE: THE FACTORIAL The factorial of a positive integer "N" is defined as:

$$N! = N * (N-1) * (N-2) ... * 3 * 2 * 1$$

Thus the factorial of 4 is 4 * 3 * 2 * 1, and the factorial of 3 is 3 * 2 * 1. You can then express the following relationships for the factorial:

4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1

In general, you can write the following Pascal statement to calculate the factorial using a counter:

```
Factorial := Factorial * Kounter;
```

The variable "Kounter" can be incremented from 1 to N or decremented from N to 1. The following program uses this logic in a loop with a decremented step.

```
{ ------------------------------ figure 4-5 ------------------------------ }
PROGRAM FactorialProg1(INPUT,OUTPUT);
VAR
  Factorial       :REAL;
  Kounter, Number :INTEGER;
BEGIN
  WRITE('Give me a number, and I will tell you the factorial: ');
  READLN(Number);
  Factorial := 1;
  FOR kounter := Number DOWNTO 1 DO
    Factorial := Factorial * Kounter;
  WRITELN('The factorial of ', Number,' is ', Factorial:0:0);
  WRITELN;
  WRITELN('Press ENTER to continue..');
  READLN
END.
```

Notice that the variable "Factorial" must be initialized to the value 1 before starting the iterative process. A sample run of the program gives the following:

```
Give me a number, and I will tell you the factorial: 8
The factorial of 8 is 40320

Press ENTER to continue..
```

TIP Although the factorial of a number is always an integer, using the type REAL (or the Turbo Pascal type LONGINT) for the variable "Factorial" gives you a large storage size with which to receive the quickly increasing results of factorial calculations. If you use the INTEGER type, the program will start giving you funny results after "7!"

DRILL 4-2

Modify the previous program to test the input value of the number. If the value is zero, the program should exit without going through the loop. You may use a GOTO statement or the Turbo Pascal function EXIT.

4-4 NESTED LOOPS

Like any other statement, the FOR loop statement can be used inside another loop. In this case it is said that the *inner loop* is nested inside the outer loop. You can nest as many loops as you wish inside one another, according to your application. The next program displays on your screen the following array of numbers.

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

The array consists of three rows and five columns. You can control the number of rows and columns by using the counters of two nested loops. As you can see, for each round of the outer loop counter (Row), the inner loop counter (Column) loops five times. The values that appear in the output are the values of the counter "Column." Notice that a blank line is displayed after a complete row is done, using the outer loop counter.

```
{------------------------------ figure 4-6 ------------------------------}
PROGRAM NestedLoops(OUTPUT);
VAR
   Row, Column :INTEGER;
BEGIN
   FOR Row := 1 TO 3 DO  { Start of the outer loop }
     BEGIN
        FOR Column := 1 to 5 DO{ Start of the inner loop }
           WRITE(Column, '  ');     { End of the inner loop }
     WRITELN  { This statement belongs to the outer loop }
     END                         { The end of the outer loop }
END.
```

TIP Notice the two END keywords in the previous program. The first one comes without a semicolon because it is the last statement in the main block (the program main body). Also, the keyword WRITELN, which comes before this END, was not terminated by a semicolon. This is because it is the last

statement in the loop block. All of these are options, but you may use the semicolons if you wish. If you add another statement at the end of the program (to suspend the screen, for instance), the situation will change.

DRILL 4-3

Modify the last program to draw the fifty stars of the American flag, as shown:

```
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
```

4-5 THE WHILE LOOP

The WHILE loop construct contains the necessary condition to terminate the loop, but unlike the FOR loop, no counter is included. It takes the general form:

WHILE condition **DO**
statement;

This form simply says: "Execute the following statement as long as the condition is TRUE."

When the loop is entered, the condition (a boolean expression) is evaluated. If it is TRUE, the statement that follows the keyword DO is executed. The loop will be repeated, and the statement will be re-executed until the condition becomes FALSE. In your program, you must include the necessary logic to make the condition FALSE at the right time. You may use a counter with this loop, but you need to increment or decrement the counter yourself.

The following program demonstrates the same algorithm of calculating the average of a set of numbers entered from the keyboard but uses the WHILE loop. The condition is here used to test the value of a counter "Kounter" against the maximum number of elements "N." When this maximum is reached the loop exits.

```
{ ---------------------------- figure 4-7 ---------------------------- }
PROGRAM AverageProg2(INPUT,OUTPUT);
VAR
  Average, Sum, Number :REAL;
```

```
   Kounter, N           :INTEGER;
BEGIN
   Sum := 0;
   Kounter := 1;
   WRITE('Enter Number of Elements:');
   READLN(N);
   WHILE Kounter <= N DO
      BEGIN
         WRITE('Enter Element #',kounter,': ');
         READLN(Number);
         Sum := Sum + Number;
         Kounter := Kounter + 1
      END;
   Average := Sum / N;
   WRITELN;
   WRITELN('Sum of Numbers = ', Sum:0:2);
   WRITELN('Average of Numbers = ', Average:0:2);
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

Notice that the counter is initialized at the beginning of the program and incremented inside the loop. The initial value is used for the first round in the loop (Kounter := 1), because the incrementing takes place after the process. This is one way to do it, but other arrangements are used in the next few programs. Notice also that when you want to include more than one statement in the WHILE loop, you must use the BEGIN-END blocks.

The following is a sample run of the program:

```
Enter Number of Elements: 3
Enter Element #1: 1
Enter Element #2: 2
Enter Element #3: 3

Sum of Numbers = 6.00
Average of Numbers = 2.00

Press ENTER to continue..
```

If you do not want to enter the number of elements beforehand, you can count them inside the loop. In this case you need a clue to end the loop, such as entering a negative number. Look at this modified version of the program, where the input number is tested with every round to see if it is –1.

```
{ ------------------------------ figure 4-8 ------------------------------ }
PROGRAM AverageProg3(INPUT,OUTPUT);
VAR
   Average, Sum, Number :REAL;
   Kounter              :INTEGER;
BEGIN
   Sum := 0;
   Average := 0;
   Number := 0;
   Kounter := 0;
   WHILE Number <> -1 DO
     BEGIN
       Kounter := Kounter + 1;
       Sum := Sum + Number;
       WRITE('Enter element #',kounter,' (or -1 to end): ');
       READLN(Number)
     END;
   IF Kounter > 1 THEN
     Average := Sum / (Kounter - 1);
   WRITELN;
   WRITELN('Sum of Numbers = ', Sum:0:2);
   WRITELN('Average of Numbers = ', Average:0:2);
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following is a sample run:

```
Enter element #1 (or -1 to end): 1
Enter element #2 (or -1 to end): 2
Enter element #3 (or -1 to end): 3
Enter element #4 (or -1 to end): -1

Sum of Numbers = 6.00
Average of Numbers = 2.00

Press ENTER to continue..
```

Notice the following points in this program:

- The input statement comes at the end of the loop block so that the input value can be tested before any processing.
- The Average is calculated by dividing the sum by the value of the counter decremented by one. This is to counteract the extra round which took place when the Number was -1.

- The Average is calculated only if the variable "Kounter" is not equal to 1. This is to avoid the "divide by zero" error, in case you want to exit the program without entering any data. In such a case you would get the following response:

```
Enter element #1 (or -1 to end): -1
Sum of Numbers = 0.00
Average of Numbers = 0.00

Press ENTER to continue..
```

DRILL 4-4

Use the WHILE loop construct to write a program to display a multiplication table as in the following example:

```
1 * X = Y
2 * X = Y
3 * X = Y
4 * X = Y
5 * X = Y
6 * X = Y
7 * X = Y
8 * X = Y
9 * X = Y
...
```

The value of X is received from the keyboard and the value Y is the multiplication result.

4-6 THE REPEAT LOOP

This loop is used to execute a group of statements until a specified condition is met. It takes the form:

```
REPEAT
   statement-1;
   statement-2;
   ...
   statement-n;
UNTIL condition;
```

As you can see in the form, this loop is ready to execute more than one statement without using the BEGIN-END blocks. Another difference between the WHILE loop and the REPEAT loop is that the REPEAT loop is executed at least once, regardless of the condition, because it starts each round by executing the statements and ends by testing the condition. In some applications this feature is necessary.

Look at the factorial algorithm using a REPEAT loop:

```
 ...
   Factorial := 1;
   Kounter := Number;
REPEAT
   Factorial := Factorial * Kounter;
   Kounter := Kounter - 1;
UNTIL Kounter = 0;
```

When the "Kounter" reaches zero (which means that the value "1" was already used up), no other rounds are needed, and the loop is terminated. You may also use the stepping-up algorithm, thus:

```
...
Factorial := 1;
Kounter := 1;
REPEAT
   Factorial := Factorial * Kounter;
   Kounter := Kounter + 1;
UNTIL Kounter = Number + 1;
```

In this case the loop is terminated when the value of Kounter reaches Number+1, which means that the value Number was already used up.

In the following program this REPEAT loop is nested in a WHILE loop. The program will be repeatedly executed until you enter 0 to terminate it.

```
{ ------------------------------- figure 4-9 ---------------------------- }
PROGRAM FactorialProg2(INPUT,OUTPUT);
VAR
  Factorial       :REAL;
  Kounter, Number :INTEGER;
BEGIN
  WRITE('Give me a number (or 0 to exit): ');
  READLN(Number);
  WHILE Number <> 0 DO              { Start of the WHILE loop }
    BEGIN
      Factorial := 1;
```

```
        Kounter := 1;
        REPEAT                         { Start of the REPEAT loop }
           Factorial := Factorial * Kounter;
           Kounter := Kounter + 1;
        UNTIL Kounter = Number + 1;    { End of the REPEAT loop }
        WRITELN('The factorial of ', Number,' is ', Factorial:0:0);
        WRITE('Give me a number (or 0 to exit): ');
        READLN(Number)
     END;                              { End of the WHILE loop }
   WRITELN('I am out of here!')
END.
```

Notice here that two similar input statements are used, one before the WHILE loop and one inside it. The first one is used to initialize the variable "Kounter" before entering the loop, in order to be ready for testing within the loop.

A sample run of the program gives the following:

```
Give me a number (or 0 to exit): 3
The factorial of 3 is 6
Give me a number (or 0 to exit): 5
The factorial of 5 is 120
Give me a number (or 0 to exit): 0
I am out of here!
```

DRILL 4-5

Rewrite the last program using an inner FOR loop and an outer WHILE loop.

SUMMARY

In this chapter you were introduced to three control structures used to build loops. These structures are:

- The FOR loop
- The WHILE loop
- The REPEAT loop

1. The FOR is used to repeat a statement or a block of statements a specified number of times. The loop takes the general form:

   FOR control-variable := expression-1 **TO** expression-2 **DO**
      statement;

where:

control-variable	is the loop counter,
expression-1	is the initial value, and
expression-2	is the final value.

2. An alternate form of the FOR loop is used to decrement the counter:

```
FOR control-variable := expression-1
        DOWNTO expression-2 DO
   statement;
```

3. The WHILE loop is used to execute a statement or a block of statements as long as a specified condition is TRUE. The construct takes the general form:

```
WHILE condition DO
   statement;
```

4. With both the FOR and the WHILE loops you can use multiple statements by including them in a BEGIN-END block.
5. The REPEAT loop is used to execute a group of statements until the specified condition fails. It takes the general form:

```
REPEAT
   statement-1;
   statement-2;
   ...
   statement-n;
UNTIL condition;
```

6. You understand now that the main difference between the REPEAT loop and the other two is that the statements inside the REPEAT loop are executed at least once regardless of the condition.
7. You understand also that the REPEAT loop can handle many statements without using BEGIN-END blocks.
8. Finally, you learned in this chapter that loop constructs may be nested inside other constructs (including other loops).

CHAPTER FIVE

DATA ARCHITECTURE

5-1 ORDINAL DATA TYPES

The data types explained so far are all predefined in the language and are called simple data types, as opposed to ***structured*** data types. Each datum of a simple data type is one single element, while in structured types (such as arrays) a datum may contain a collection of items.

Simple types fall into two main categories:

- The ordinal type
- The real type

The ordinal types include the INTEGER, CHAR, and BOOLEAN types. An ordinal type is distinguished by data values that form a series of discrete elements such that every element has a discrete predecessor (except the first element) and successor (except the last element). Integers are like that, as they form a set of distinct numbers ranging from –(MAXINT+1) to +MAXINT. The element "4," for example, is preceded by "3" and followed by "5." The type CHAR includes a set of characters ordered sequentially according to their ordinal numbers. The type BOOLEAN contains the set "TRUE and FALSE." The value FALSE has the ordinal number "0" while TRUE has the ordinal number "1."

Real numbers, on the other hand, are not discrete. For example, between the number "0" and "1" there exists an infinite number of fractions. Between any two real numbers, then, there is another real number.

ENUMERATIONS It is sometimes useful in a program to define days of the week as integers in order to make the program code more readable. In this

case, you need to either assign each day a number or to declare each a named constant as in:

```
CONST
   Monday = 0;
   Tuesday = 1;
   Wednesday = 2;
   Thursday = 3;
   Friday = 4;
   Saturday = 5;
   Sunday = 6;
```

After these declarations you can refer to any of these days by name:

```
IF Today = Sunday THEN
   WRITELN('Sorry, we are closed on Sundays..');
```

In this statement an integer variable "Today" is tested to check if it is "Sunday"; in other words, if it contains the value "6." Using such declarations will take a lot of programming effort, though, especially when you have a large number of constants (such as the names of the months).

The *enumerated* type gives you a shortcut to doing the same thing. Look at the following declaration:

```
VAR
   Day :(Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
```

In this declaration, the identifiers representing the months are listed in an ordered series and separated by commas. Thus, Monday is internally coded as 0 and Sunday is coded as 6. Other days are represented by numbers between 0 and 6 according to their sequence in the enumeration. It is, however, illegal to read or write these values directly as you do with simple types (using WRITELN and READLN statements). With enumerations you may use any of the following operations:

A. You may assign any one of the enumeration elements to the variable "Day" like this:

```
Day := Friday;
```

but it is illegal to assign an explicit number to the variable "Day," such as "Day := 1." This feature assures that the enumeration will be only assigned valid data.

B. You can obtain and use the values associated with the enumeration elements using the ORD function. For example:

```
WRITELN(ORD(Monday));        gives the value "0"
WRITELN(ORD(Tuesday));       gives the value "1"
```

C. You may also use the functions PRED and SUCC to obtain the predecessor and the successor of a specified element:

```
WRITELN(PRED(Friday));       gives the value "3"
WRITELN(SUCC(Monday));       gives the value "1"
```

D. You can compare values of the enumerated type using the boolean operators (simple or compound), like this:

```
IF (Day = Saturday) OR (Day = Sunday) THEN
   WRITELN('This is a weekend day.');
```

Again, you cannot use the explicit values in comparisons such as "IF Day = 2." This results in an error.

In the following program a FOR loop uses the enumeration "Month" to display the corresponding integer values from 0 to 11.

```
{ ---------------------------- figure 5-1 ---------------------------- }
PROGRAM Enumeration1(OUTPUT);
VAR
  Month :(Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
BEGIN
  WRITELN;
  FOR Month := Jan TO Dec DO
    WRITE(ORD(Month),' ')
END.
```

The output of this program is:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

As you can see in the output, the values that correspond to the twelve months range from 0 to 11. If you would like to see the values range from 1 to 12 as the months in the calendar do, you can use the expression "ORD(month)+1" instead of the expression "ORD(month)."

The enumerated type is an ordinal data type and is classified as a user-defined type.

SUBRANGES The *subrange*, another user-defined ordinal type, helps to eliminate out-of-range data. For example, instead of using the INTEGER type

to represent the month numbers, you can declare the variable "Month" as a subrange like this:

```
VAR
  Month : 1..12;
```

As such, any value outside the range 1 to 12 will be considered an error either in compilation or at runtime. In other words, you cannot, after this declaration, write a statement like this in your program:

```
Month := 13;          ----> illegal statement
```

Also, if a user responds to an input statement by entering an out-of-range number, the program will issue the proper error message, though with some compilers you have to set a switch to make the compiler detect out-of-range errors.

The type used to represent month values in this example is INTEGER. It is called the *base type* of the subrange. You may use any ordinal type as the base type. For example, you can declare the uppercase letters as a subrange using the base type CHAR as follows:

```
VAR
  Uppercase : 'A'..'Z'
```

In this case, only the uppercase letters will be permitted as data for the subrange "Uppercase."

The following example demonstrates the use of a subrange to represent months, followed by a CASE statement to classify months as seasons. The program prompts you to enter the month number and displays the season this month belongs to.

```
{ ----------------------------- figure 5-2 ----------------------------- }
PROGRAM Subrange1(INPUT,OUTPUT);
VAR
  MonthNumber :1..12;
BEGIN
  WRITE('Please enter the number of the month: ');
  READLN(MonthNumber);
  CASE MonthNumber OF
    12, 1 ,2  :WRITELN('This is wintertime.');
    3, 4, 5   :WRITELN('This is springtime.');
    6, 7, 8   :WRITELN('This is summertime.');
```

```
    9, 10, 11 :WRITELN('This is autumn.')
  END
END.
```

The following are two sample runs. The second one gave a runtime error message because the number "14" was entered as a month number:

RUN 1:

```
Please enter the number of the month:2
This is wintertime.
```

RUN 2:

```
Please enter the number of the month: 14
Runtime error 201 at 0000:00BE.
```

The subrange, in general, can be a subset of any previously defined sequence (of the ordinal type). So, if the enumeration "Day" has already been defined in your program, you may then define a subrange like this:

```
VAR
  WorkingDay : Monday..Friday;
```

This is valid because the words "Monday" and "Friday" are already known to the compiler.

There are some restrictions on using enumerations and subranges:

A. The first element in a subrange must be less than the last one.

B. Though a subrange can be a subset of an enumeration, an enumeration cannot use elements from another enumeration.

C. The enumeration elements cannot be used as identifiers for other variables. It is the same as declaring the same variable identifier twice in one program.

DRILL 5-1

Write a declaration to define the following subranges:

A. The uppercase letters

B. The lowercase letters

C. The decimal digits

Accept values that correspond to each subrange and display them preceded by the proper message. The output may look something like this:

```
Lowercase letter     : r
Uppercase letter     : T
Digit                : 5
```

5-2 THE TYPE SECTION

The enumerations and subranges are usually associated with the TYPE statement, which is used to declare new user-defined types or to rename predefined types. The TYPE statement comes in the TYPE section of the declaration part. It takes the form:

```
TYPE
   type-name = type-definition;
```

where "type-name" is the type identifier, and "type-definition" is a predefined type or new type definition.

RENAMING TYPES It is possible to rename any data type, even the simple types such as INTEGER, as in this example:

```
TYPE
   Day = INTEGER;
```

In this declaration the type INTEGER is given a new name (Day). Thus, in the VAR section, you can declare some other variables of the type "Day" like this:

```
VAR
   Holiday, Yesterday, Tomorrow : Day;
```

The type "Day" is actually the type INTEGER, but given another name (a synonym). In your program, you may use either one of the two names (INTEGER or Day) because the type INTEGER is still recognized by the compiler. This is not, however, the proper use of the TYPE statement. It is meant to be used for naming types such as enumerations and subranges.

NAMING USER-DEFINED TYPES Instead of declaring enumerations and subranges in the VAR section, it would be better to declare them as types. Look at these declarations:

```
TYPE
   Day = (Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);
   WorkingDay = Monday..Friday;
```

Here, two new types are declared: the enumerated type "Day," and the subrange "WorkingDay." Notice that the subrange is defined as a subset of the enumeration "Day." Needless to say, the enumeration declaration must come first in this case.

You can use these new types in the VAR section to declare variables in the same way you use the predefined types of the language. Thus:

```
VAR
   Today, Yesterday, Tomorrow, Holiday :Day;
   DayOff :WorkingDay;
```

The use of the TYPE statement saves you the effort of writing long declarations for the enumeration variables "Today," "Yesterday," "Tomorrow," and "Holiday." They are all simply of the type "Day."

Now in your program you may write assignment statements like the following:

```
Holiday := Friday;
DayOff := Tuesday;
Tomorrow := Sunday;
```

In order to see the values contained in your variables, use an output statement such as:

```
WRITELN(ORD(Holiday), ', ',ORD(DayOff),', ', ORD(Tomorrow));
```

In this case, the statement will give you the values 4, 1, and 6 respectively.

In standard Pascal the TYPE section should come in the following sequence relative to the other sections:

LABEL section
CONST section
TYPE section
VAR section

In Turbo Pascal, as mentioned before, the order is not important, but the TYPE section should still precede the VAR section because it contains the definitions of the user-defined types.

DRILL 5-2

Which of the following declarations are valid if they all come in one program?

```
     TYPE
{1}   Football    = (Saints, Cowboys);
{2}   Games       = (Football, Baseball, Basketball)
```

```
{3}   Week        = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
{4}   Weekend     = Sat..Sun;
{5}   Compiler    = (C, Pascal, Fortran, Ada, Basic);
      VAR
{6}   WholeWeek   :Week;
{7}   WorkingDay  :(Mon, Tue, Wed, Fri);
{8}   Weekday     :Mon..Fri;
{9}   SW          :(Compiler, OperatingSystem, ApplicationProgram);
{10}  DpTools     :(Hardware, Software, PeopleWare);
{11}  DpTool      :(HW, SW, PW);
{12}  C           :(TurboC, QuickC);
{13}  Margin      : -10..+10;
```

5-3 ARRAYS AS DATA STRUCTURES

If you would like to represent the names of the players on a football team using only simple data types, you would need to use one variable for each player's name. In such a case, you would need too many variables, such as:

```
FirstPlayer
SecondPlayer
ThirdPlayer
...
```

This is not a good idea. Now imagine the case if you were dealing with a class of one hundred students. It would be almost impossible to use one hundred variables to store names.

The practical way to store this kind of data is to use the array data structure, which is useful for storing a collection of related data items. In the case of the football team you would need to declare only one *subscripted* variable, and you would represent your data like this:

```
Player[1]
Player[2]
Player[3]
...
```

The name of the variable is "Player," and the number between the brackets is called the *subscript* or *index*. Changing the index gives you a new memory location in which to store a new name. This type of data structure is called *one-dimensional array*. It is useful to represent data such as names of a group of people, scores of one student in several classes, or any similar set of related items (see table 5-1).

Table 5-1 Example of a one-dimensional array

Player[1]	Player[2]	Player[3]	Player[4]	Player[5]	
Able	Baker	Charlie	John	Sam	...

In chapter two you met a special type of one-dimensional array (the PACKED ARRAY OF CHAR), which is used to store a string of text in standard Pascal, and you already know that each element (character) in this array is referred to by a number (index).

In other applications you may need a *two-dimensional array*, which is capable of handling more complicated structures. For example, suppose that you want to store the scores of a group of students in different classes, as represented in table 5-2.

Each element in this table is related to a row (the student number) and a column (the class number); these are the two dimensions of the array. The data item itself is a real number.

Table 5-2 Example of a two-dimensional array

	Class # (second index)				
	1	2	3	4	5
Student # (first index)					
1	55.5	60.9	66.5	80.3	70.5
2	89.1	77.6	99.9	88.7	50.3
3	40.5	67.4	90.5	45.1	66.9
...	...	...	...	...	...
100	68.8	87.2	90.4	60.1	60.4

To represent the data in this table your variables will look something like this:

```
StudentScore[3][4]
```

This variable represents the score of student #3 in class #4; in other words, the number at the intersection of row #3 and column #4. You may assign a numeric value which represents a score to this variable, thus:

```
StudentScore[3][4] := 45.1;
```

Compare now the following assignment statements to the values in the table:

```
StudentScore[1][1] := 55.5;      { the score of student #1 in class #1 }
StudentScore[1][2] := 60.9;      { the score of student #1 in class #2 }
StudentScore[3][5] := 66.9;      { the score of student #3 in class #5 }
StudentScore[100][2] := 87.2;    { the score of student #100 in class #2 }
```

Arrays are classified as structured data types (as opposed to the simple [or *unstructured*] types you have used thus far). There are many other structured data types in Pascal which are useful for different applications.

As a matter of fact, the quality of a program is mainly measured by two criteria:

A. The structural efficiency of the program; that is, how readable, easy to debug, and prone to errors it is

B. The use of the most efficient data structures applicable, to save time and enable the program to manipulate data in the most efficient way

NOTE ON TERMINOLOGY

An array variable may be called either a subscripted variable or an *indexed* variable. The array elements referred to by the array variables are also called array *components*. In mathematics, a one-dimensional array is called a *vector*, while a two-dimensional array is called a *matrix*. You may come across these names in mathematical applications.

5-4 ONE-DIMENSIONAL ARRAYS

A one-dimensional array is declared using the following form:

VAR
array-name : **ARRAY**[index-range] **OF** element-type;

If you want, for example, to declare an array to store test scores of ten students as real numbers, you can declare your array like this:

```
VAR
   Score : ARRAY[1..10] OF REAL;
```

This array (named "Score") can hold up to 10 real numbers. The index range [1..10] indicates that the indexes of the array elements start from 1 and end at 10. The index range, which is a subrange (of integers in this example), can be of any ordinal type, but the array elements can be of any data type. The above declaration, then, reserves a sequence of ten memory locations in which to store ten REAL values of ten array elements.

EXAMPLE: SCORES OF ONE STUDENT In the following program the array "Score" is used to store the scores of one student in six different classes. The scores are entered from the keyboard, then the sum and average of the scores are displayed.

```
{ ----------------------------- figure 5-3 ----------------------------- }
PROGRAM Scores1(INPUT,OUTPUT);
CONST
   NumberOfClasses = 6;
VAR
   Score :ARRAY[1..NumberOfClasses] OF REAL;
   Average, SumOfScores :REAL;
   Index                :INTEGER;
BEGIN
{ Read the scores array }
{ --------------------- }
   FOR Index := 1 TO NumberOfClasses DO
      BEGIN
         WRITE('Enter score for class #', Index,': ');
         READLN(Score[Index])
      END;
{ Calculate the sum }
{ ----------------- }
   SumOfScores := 0;
   FOR Index := 1 TO NumberOfClasses DO
      SumOfScores := SumOfScores + Score[Index];
{ Calculate the average }
{ --------------------- }
   Average := SumOfScores / NumberOfClasses;
{ Display Results }
{ --------------- }
   WRITELN;
   WRITELN('Sum of scores = ', SumOfScores:0:2);
   WRITELN('Average of scores = ', Average:0:2);
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

A sample run of the program gives the following:

```
Enter score for class #1: 90
Enter score for class #2: 80
Enter score for class #3: 85
Enter score for class #4: 75
```

```
Enter score for class #5: 89
Enter score for class #6: 91

Sum of scores = 510.00
Average of scores = 85.00

Press ENTER to continue..
```

The following points in this program are worthy of your attention:

A. The size of the array is declared as a constant (NumberOfClasses).

B. The index-range of the array is declared using the previously defined constant "NumberOfClasses" as follows:

```
Score :ARRAY[1..NumberOfClasses] OF REAL;
```

This is the same as:

```
Score :ARRAY[1..6] OF REAL;
```

The first declaration, however, is much better, because if you would like to process a different number of classes, you just change the value of the constant "NumberOfClasses" without modifying the program main body.

C. Notice that after the program reads the scores, they are stored in the array elements and are available in memory. This means that the sum can be processed later in the program. When you calculated the sum and the average of some numbers before (program 4-4), you had to accumulate the values during data entry in one variable "Sum." Now, you have six variables.

D. The index of the array is used as a counter in the FOR loops, both for reading data and calculating the sum. Actually, the index of the array is very useful for displaying results, especially if you like to display the results in table form.

DISPLAYING TABULATED RESULTS The following program deals with the same problem but displays the results in a tabulated form.

```
{ ----------------------------- figure 5-4 ----------------------------- }
PROGRAM Scores2(INPUT,OUTPUT);
CONST
  NumberOfClasses = 6;
  Tab = '         ';               { 9 spaces }
VAR
  Score :ARRAY[1..NumberOfClasses] OF REAL;
  Average, SumOfScores :REAL;
```

```
   Index                  :INTEGER;
BEGIN
{ Read the scores array }
{ --------------------- }
   FOR Index := 1 TO NumberOfClasses DO
      BEGIN
         WRITE('Enter score for class #', Index,': ');
         READLN(Score[Index])
      END;
{ Calculate the sum }
{ ----------------- }
   SumOfScores := 0;
   FOR Index := 1 TO NumberOfClasses DO
      SumOfScores := SumOfScores + Score[Index];
{ Calculate the average }
{ --------------------- }
   Average := SumOfScores / NumberOfClasses;
{ Display Results }
{ --------------- }
   WRITELN;
   WRITELN(Tab,'CLASS #');
   WRITE('      ');                  { 6 spaces }
   FOR Index := 1 TO NumberOfClasses DO
      WRITE(Index:7);
   WRITELN;
   WRITE(Tab);
   FOR Index := 1 TO NumberOfClasses DO
      WRITE('-------');
   WRITELN;
   WRITE('SCORES  ');
   FOR Index := 1 TO NumberOfClasses DO
      WRITE(Score[Index]:7:2);
   WRITELN;
   WRITE(Tab);
   FOR Index := 1 TO NumberOfClasses DO
      WRITE('-------');
   WRITELN;
   WRITELN(Tab,'Sum of scores = ', SumOfScores:0:2);
   WRITELN(Tab,'Average of scores = ', Average:0:2);
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

This is a sample run:

```
Enter score for class #1: 90.5
Enter score for class #2: 80.5
Enter score for class #3: 86.2
Enter score for class #4: 90.3
Enter score for class #5: 74.8
Enter score for class #6: 98.5

         CLASS #
            1      2      3      4      5      6
         ------------------------------------------
SCORES     90.50  80.50  86.20  90.30  74.80  98.50
         ------------------------------------------
         Sum of scores = 520.80
         Average of scores = 86.80
Press ENTER to continue..
```

In this program extensive use of loops has been made to display the dashed lines, the class numbers, and the scores; this makes the program more generic. For example, the dashed line could be displayed using the statement:

```
WRITELN('         ------------------------------------------');
```

This is useful only for six classes, but the following statements:

```
WRITE(Tab);
FOR Index := 1 TO NumberOfClasses DO
   WRITE('-------');
```

are useful for any number of classes, because a seven-dash segment is displayed for each class. Thus, if you had only four classes, the output would look like this:

```
         CLASS #
            1      2      3      4
         ----------------------------
SCORES     80.00  90.00  85.00  75.00
         ----------------------------
         Sum of scores = 330.00
         Average of scores = 82.50
```

Notice that the number of dashes is equal to the field width specified in the output format of "Score" and "Index":

```
WRITE(Index:7);
WRITE(Score[Index]:7:2);
```

Note also the use of the constant "Tab" (which contains nine spaces) for proper indentation of the output.

A weak point of this program is that we have to repeat the same lines of code every time we want to draw a line. Such repetitive tasks can instead be programmed separately as *procedures* and called whenever wanted. This is discussed later in the book.

DRILL 5-3

Write a Pascal program to read and store the scores of five students in one test, then display the output as shown below:

```
Student #          Score
------------------------
  1               90.00
  2               88.00
  3               91.00
  4               78.00
  5               75.00
------------------------
Average score = 84.40
```

DECLARATION OF ARRAYS IN THE TYPE SECTION It is preferable that array declarations be associated with the TYPE statement, as in this example:

```
TYPE
   AnArray = ARRAY[1..6] OF INTEGER;
VAR
   MyArray :AnArray;
```

In this case you can declare more than one array of the type "AnArray" in the VAR section:

```
VAR
   YourArray, MyArray :AnArray;
```

It is also possible to use a previously declared subrange as an index range for an array, like this:

```
TYPE
   MyRange = 1..6;
   AnArray = ARRAY[MyRange] OF INTEGER;
VAR
   MyArray :AnArray;
```

Although we have started our arrays from the index "1," there is no obligation to do so. The index range can be any valid subrange, but you must always remember not to exceed the defined index range.

EXAMPLE: SORTING AN ARRAY If you would like to sort some numbers (or names), the best way is to store them in an array, then use one of the sorting algorithms. A simple way (but not the most efficient) to sort numbers in an ascending order is known as the "bubble sort." The algorithm is as follows:

1. Compare the first element to the second one. If the first element is greater, swap them.
2. Repeat the comparison between the first element and each of the rest of the array elements. If it is greater than any element, swap them.
3. By the end of these comparisons the first element will be the smallest in the array.
4. Repeat the previous steps for the second element, the third, and so on until the next-to-last element.

After this process is completed, the array will be sorted in an ascending order. This algorithm is demonstrated in the following program. The comparisons need two nested loops. The outer loop (index "I") starts from the first element (I=1) and ends before the last element (I=ArraySize-1). The inner loop (index "J") starts one step after the start of the outer loop (J=I+1) and goes all the way to the last element (J=ArraySize).

```
{ ------------------------------ figure 5-5 ------------------------------ }
PROGRAM Sorting(INPUT,OUTPUT);
CONST
  ArraySize = 6;
TYPE
  Range = 1..ArraySize;
  NumbersArray = ARRAY[Range] OF INTEGER;
VAR
  Numbers    :NumbersArray;
  I, J , Pot :INTEGER;
BEGIN
{ Read the array }
{ ------------- }
  FOR I := 1 TO ArraySize DO
    BEGIN
      WRITE('Enter element #', I,': ');
      READLN(Numbers[I])
```

```
    END;
{ Sort the array }
{ -------------- }
   FOR I := 1 TO ArraySize-1 DO                 { outer loop }
      BEGIN                                     { optional block }
         FOR J := I+1 TO ArraySize DO           { inner loop }
            BEGIN
               IF Numbers[I] > Numbers[J] THEN
                  BEGIN                         { swap contents }
                     Pot := Numbers[J];
                     Numbers[J] := Numbers[I];
                     Numbers[I] := Pot
                  END
            END                                 { end of inner loop }
      END;                                      { end of outer loop }
{ Display Results }
{ --------------- }
   WRITELN;
   WRITELN('The sorted array is:');
   FOR I := 1 TO ArraySize DO
      WRITELN(Numbers[I]);
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

A sample run is as follows:

```
Enter element #1: 6
Enter element #2: 33
Enter element #3: 4
Enter element #4: 2
Enter element #5: 55
Enter element #6: 9

The sorted array is:
2
4
6
9
33
55
Press ENTER to continue..
```

Swapping the contents of two elements is done by using a third variable (Pot) to hold the contents of one variable temporarily, thus:

```
Pot := Numbers[J];
Numbers[J] := Numbers[I];
Numbers[I] := Pot
```

This process is similar to swapping the contents of two cups, one of which contains coffee and the other tea; all you need is an empty cup (Pot).

To have the array sorted in a descending order, just reverse the "greater than" process to "less than."

DRILL 5-4

Modify your program from drill 5-3 to display the best score and the number of the highest scoring student in the class. The output should look like this:

```
Student #          Score
------------------------
   1               70.00
   2               88.00
   3               67.00
   4               90.00
   5               86.00
------------------------
Average score  = 80.20
The best score = 90.00
The best of the class is student #4
```

You may use the following algorithm to obtain the highest number in the array of scores:

1. Store the score of the first student in a variable such as "BestScore," and the index of that student in a variable "BestOfClass."
2. Starting from the second element in the array of scores, and continuing all the way to the end, repeat the following comparison:

   If any number is greater than "BestScore," store it in "BestScore" and store its index in "BestOfClass."
3. By the end of the loop, the variable "BestScore" will contain the highest score, and the corresponding student number will be stored in "BestOfClass."

5-5 TWO-DIMENSIONAL ARRAYS

To declare a two-dimensional array, use the form:

```
VAR
array-name : ARRAY[index-range-1, index-range-2]
                   OF element-type;
```

You may also declare it in the type section as follows:

```
TYPE
type-name = ARRAY[index-range-1, index-range-2]
                   OF element-type;
```

where index-range1 and index-range2 are the ranges of the first and second dimensions.

Look at this declaration:

```
TYPE
   Score = ARRAY[1..100, 1..6] OF INTEGER;
```

This statement declares an array "Score," which can store the scores of 100 students in 6 different classes; generally speaking, it can store up to 600 integers. As you can see, each dimension is represented by a subrange.

You can also declare a multidimensional array of any number of dimensions using the general form:

```
TYPE
type-name = ARRAY[index-range-1, index-range-2,
        ..., index-range-n] OF element-type;
```

In most applications, however, you will not need more than two dimensions.

EXAMPLE: SCORES OF STUDENTS The following program is used to read the scores of a number of students in different classes as represented in table 5-2. For simplicity of demonstration, only four students and three classes will be considered; you can, however, modify the number of students or classes by simply changing the values of the two constants "NumberOfClasses" and "NumberOfStudents."

```
{ ------------------------------ figure 5-6 ------------------------------ }
PROGRAM Scores3(INPUT,OUTPUT);
{ using two-dimensional array }
CONST
```

```
   NumberOfClasses = 3;          { Change this number for more classes }
   NumberOfStudents = 4;         { Change this number for more students }
   Tab = '       ';              { 7 spaces }
   Dash = '-';
   NumberOfDashes = 23;
TYPE
   ScoreArray = ARRAY[1..NumberOfStudents, 1..NumberOfClasses] OF REAL;
   AverageArray = ARRAY[1..NumberOfStudents] OF REAL;
VAR
   Score                                  :ScoreArray;
   Average                                :AverageArray;
   SumOfScores                            :REAL;
   StudentCount, ScoreCount, DashCount    :INTEGER;
BEGIN
{ Read the scores array }
{ --------------------- }
   FOR StudentCount := 1 TO NumberOfStudents DO
      BEGIN
         WRITELN;
         WRITELN('Scores of student #', StudentCount,': ');
         FOR ScoreCount := 1 TO NumberOfClasses DO
            BEGIN
               WRITE('Enter score for class #', ScoreCount,': ');
               READLN(Score[StudentCount, ScoreCount])
            END;
      END;
{ Calculate the average for each student }
{ -------------------------------------- }
   FOR StudentCount := 1 TO NumberOfStudents DO
      BEGIN
         SumOfScores := 0;  { Initialize for each student }
         FOR ScoreCount := 1 TO NumberOfClasses DO
            SumOfScores := SumOfScores + Score[StudentCount, ScoreCount];
         Average[StudentCount] := SumOfScores/NumberOfClasses
      END;
{ Display results }
{ --------------- }
   WRITELN;
   WRITELN(Tab, 'Student #', Tab, 'Average');
   WRITE(Tab);
   FOR DashCount := 1 TO NumberOfDashes DO
      WRITE(Dash);
   WRITELN;
   FOR StudentCount := 1 TO NumberOfStudents DO
```

```
      WRITELN(Tab, StudentCount:3, Tab, Average[StudentCount]:12:2);
   WRITE(Tab);
   FOR DashCount := 1 TO NumberOfDashes DO
      WRITE(Dash);
   WRITELN;
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The following is a sample run:

```
Scores of student #1:

Enter score for class #1: 90
Enter score for class #2: 89
Enter score for class #3: 93

Scores of student #2:

Enter score for class #1: 80
Enter score for class #2: 70
Enter score for class #3: 60

Scores of student #3:

Enter score for class #1: 77
Enter score for class #2: 78
Enter score for class #3: 90

Scores of student #4:

Enter score for class #1: 91
Enter score for class #2: 94
Enter score for class #3: 95

        Student #       Average
        -----------------------
          1               90.67
          2               70.00
          3               81.67
          4               93.33
        -----------------------
Press ENTER to continue..
```

Notice the following in this program:

A. Two types of arrays were declared in the TYPE section, a two-dimensional array "ScoreArray" and a one-dimensional array "AverageArray." These type identifiers are used in the VAR section to declare the two arrays "Score" and "Average." The first array is used to store the scores of the four students in three classes, while the second is used to hold the averages of the four students (which are, of course, only four values).

B. Data are read through two loops, using the index "StudentCount" as a counter of students in the outer loop and "ScoreCount" as a counter of scores in the inner loop. Each value read from the keyboard is assigned to the general array variable:

```
Score[StudentCount, ScoreCount]
```

The exact location of the array element is determined by the two indexes "StudentCount" and "ScoreCount."

C. The average of scores is calculated for each student and stored in the array variable:

```
Average[StudentCount]
```

The index "StudentCount" indicates which student has each average.

D. Notice the initialization of the variable "SumOfScores" before the average calculation. This is a very important step because if it is not done, the average of the previous student will remain in the variable and be added to the new average.

ARRAY INITIALIZATION If you are assigning values to only some of the elements of an uninitialized array, do not expect that the rest of the elements will contain zeros. In such applications you have to initialize the whole array using a loop like this.

```
FOR I := 1 TO N DO
    MyArray[I] := 0;
```

You need another loop if the array is two-dimensional:

```
FOR I := 1 TO N DO
      FOR J := 1 TO M DO
         MyArray[I,J] := 0;
```

In the last example, we assigned values to each element of the array, so there was no need for initialization.

DRILL 5-5

Modify the last program to display the students' names in descending order according to their scores, as in this example:

```
Student name    Average
----------------------
Porter, Thomas    84.00
Dalton, Jack      83.33
Dixon, Jane       83.33
Bobbin, Dale      66.67
----------------------
```

SUMMARY

In this chapter you have had a review of simple and structured data types.

1. You now know that simple data types are classified as either "real" or "ordinal" types. Of the ordinal types, you learned how to use the user-defined types, enumerations and subranges.
2. You learned how to use the TYPE statement to declare a new type or rename a predefined type. It takes the general form:

   TYPE
   type-name = type-definition;

   In standard Pascal the relative sequence of the TYPE section among the other sections in the declaration part is as follows:

   LABEL section
   CONST section
   TYPE section
   VAR section

3. You learned about the array as a predefined structured data type that may be declared either in the TYPE section or VAR section. You also learned how to declare and use both one- and two-dimensional arrays. The general form to declare an array of any number of dimensions (in the TYPE section) is:

   TYPE
   type-name = **ARRAY**[index-range-1, index-range-2,
   ..., index-range-n] **OF** element-type;

CHAPTER SIX

TEXT PROCESSING

6-1 INTRODUCTION

Thus far you have dealt mainly with numeric data values. In this chapter, you learn to use characters and strings to manipulate text data, paying special attention to input and output of characters and strings using the keyboard and the screen. These devices are treated as files; they are referred to as the standard INPUT file (the keyboard), and the standard OUTPUT file (the screen).

6-2 TIPS ON OUTPUT STATEMENTS

If you would like to display many lines of text, or display numeric results in separate lines, you can use as many WRITELN statements as the number of required lines. Another way to do this (one requiring less effort) is to use the ASCII control codes "13" (Carriage Return) and "10" (Line Feed) whenever a new line is required. You can then use one WRITE or WRITELN statement to print all of the results. With most microcomputer systems the carriage return/line feed pair is interpreted as the *End-Of-Line mark*. In the following example the control character "CHR(10)" is declared as a named constant "LF" (a common abbreviation for Line Feed), and the control character "CHR(13)" as "CR" (a common abbreviation for Carriage Return). The combination of the two characters CR and LF gives the same effect as pressing Enter.

```
{ ------------------------------ figure 6-1 ------------------------------ }
PROGRAM Display1(INPUT,OUTPUT);
CONST
   LF = CHR(10);
   CR = CHR(13);
```

```
VAR
   X, Y, Z :INTEGER;
BEGIN
   WRITE('Enter three integers: ');
   READLN(X, Y, Z);
   WRITELN('X=', X, CR, LF, 'Y=', Y, CR, LF, 'Z=', Z)
END.
```

Sample run:

```
Enter three integers: 11 22 33
X=11
Y=22
Z=33
```

If you tried this program using the LF only, you would get the following output:

```
X=11
    Y=22
        Z=33
```

Try it now using the CR only, and you will find that the last result overwrites the first two. The output will be only one line like this:

```
Z=33
```

6-3 TIPS ON INPUT STATEMENTS

When you use the input statements READ or READLN some pitfalls can occur during successive reads, especially with character input. For this reason it is important to understand how the input statements work with different types of data.

When a READ or a READLN statement is executed, values are stored in the standard INPUT file (the keyboard). The stored values are then read from this file and assigned to the variables specified in the input list. Each time you press the Enter key, an End-Of-Line mark is written to the INPUT file.

USING READLN FOR NUMERIC INPUT Assume that your input contains the following numbers:

```
123 45 678      <Enter>
```

You may imagine that the numbers are stored in the INPUT file as in the following figure:

1	2	3		4	5		6	7	8	*

The End-Of-Line mark is shown at the last location and is indicated by the asterisk (*). At the first location, there is a little arrow (called the file pointer) pointing to the beginning of the file. Consider now that these values are read by the following statement:

```
READLN(X, Y, Z);
```

After the first integer (123) is read and assigned to the variable "X," the pointer moves to the space before the second numeric value (45). The second value is then read and assigned to the variable "Y," and the pointer moves to the space before the third value. When the third value is read and assigned to "Z," all of the variables will have been assigned values and the pointer moves past the End-Of-Line mark, where the work of the READLN statement ends. If you leave more than one space between numeric values, the extra spaces will be ignored and you will still get correct results.

Suppose now that you entered a fourth value by mistake:

```
123 45 678 90    <Enter>
```

The last value (90) will be ignored by the program, as the pointer will move past the End-Of-Line mark after the three values are read, in order to be ready for a subsequent read.

1	2	3		4	5		6	7	8		9	0	*

NOTE This feature of the READLN statement is inherited from the old days when data were read from *punched cards* (each card represents a line of data). The READLN was used to read only a specific number of items and eject to the next card.

You may also enter your numeric values separated by the Enter key, in which case each numeric value will be followed by the End-Of-Line mark like this:

1	2	3	*	4	5	*	6	7	8	*

As long as the three variables have not yet been assigned values, the End-Of-Line marks between the values are treated as spaces and are thus

ignored. The pointer moves from one End-Of-Line mark to another until all of the values have been read, then the pointer moves past the end of the next End-Of-Line mark, ending the READLN statement. Try the following program (which contains two READLN statements) using the values shown in the sample runs.

```
{ ----------------------------- figure 6-2 ----------------------------- }
PROGRAM ReadLnNumbers(INPUT,OUTPUT);
CONST
   CR = CHR(13);
   LF = CHR(10);
VAR
   A, C , D, E :INTEGER;
   B           :REAL;
BEGIN
   WRITE('Enter A, B, C: ');
{ If you enter more than three values, only the first three will be read }
   READLN(A, B, C);
{ Now a subsequent READLN will start to read values after the End-Of-Line
  mark, ignoring any leftovers from the previous read }
   WRITE('Enter D, E: ');
   READLN(D, E);
   WRITELN('A=',A,', B=',B:0:2,', C=', C, CR, LF,
           'D=', D,', E= ',E)
END.
```

Sample run:

```
Enter A, B, C: 1 2 3 4 5 6       ----> Enter these values
Enter D, E: 7 8                  ----> Enter these values
A=1, B=2.00, C=3                 ----> The program response
D=7, E= 8
```

Notice that the extra values (4, 5, 6) in the first input line were ignored completely, and the second read started from the value "7," which follows the End-Of-Line mark.

DRILL 6-1

Try the last program using the following inputs and study the results:

```
1 2        <Enter>
3 4 5 6    <Enter>
7 8        <Enter>
```

USING READ FOR NUMERIC INPUT With the READ statement the reading procedure is different, because after the READ statement is done, the file pointer does not move past the End-Of-Line mark, and so any subsequent READ will start from where the previous READ left off. Replace the READLN statements in the previous program with READ statements and try the following input:

```
1 2 3 4 5 6 7    <Enter>
```

When you press Enter, the program will not pause at the second input statement because the input file contains sufficient numeric values for five variables. In this case, the program displays the following results:

```
A=1, B=2.00, C=3
D=4, E= 5
```

DRILL 6-2

Using the last program with the READ statement, try the following inputs:

1.

```
1 2          <Enter>
3 4 5 6 7 <Enter>
```

2.

```
1 2 3 4    <Enter>
5 6 7      <Enter>
```

USING READ FOR CHARACTER INPUT With character input, the input statements work in a different way. The READ statement reads successive characters from the keyboard file, including the End-Of-Line mark (which is actually two characters "CR" and "LF"), and assigns each character to the next variable in the input list. Consider the following input statement:

```
READ(C1, C2, C3, C4);
```

where C1, C2, C3, and C4 are variables of the type CHAR.

If you enter the four characters that follow:

```
ABCD
```

they will all be read and assigned to the variables, thus:

C1 contains 'A'
C2 contains 'B'

C3 contains 'C'
C4 contains 'D'.

Now consider the case of an input like this:

```
A B C D
```

The first four characters (including blank spaces) in this input will be assigned to the four variables and the rest ignored, giving the following result:

C1 contains 'A'
C2 contains ' ' (blank space)
C3 contains 'B'
C4 contains ' ' (blank space)

Run the following program and use the sample run values to see how things work. Notice that the output of the program gives you both the variables' contents and the corresponding ASCII codes, which will help you to recognize any nonprintable character such as the space, the Line-Feed, or the Carriage Return.

```
{ ----------------------------- figure 6-3 ----------------------------- }
PROGRAM CharRead1(INPUT,OUTPUT);
CONST
   LF = CHR(10);
   CR = CHR(13);
VAR
   C1, C2, C3, C4 :CHAR;
BEGIN
   WRITE('Enter four characters: ');
   READ(C1, C2, C3, C4);
   WRITELN('Your inputs have been assigned to the variables as follows:', CR, LF,
           'C1= ', C1, CR, LF,
           'C2= ', C2, CR, LF,
           'C3= ', C3, CR, LF,
           'C4= ', C4);
   WRITELN('The corresponding ASCII codes are:', CR, LF,
           ORD(C1),' ', ORD(C2),' ', ORD(C3),' ',ORD(C4))
END.
```

The following are sample runs of the program.

RUN 1:

```
Enter four characters: A BCD
Your inputs have been assigned to the variables as follows:
C1= A
C2=        { blank space }
C3= B
C4= C
The corresponding ASCII codes are:
65 32 66 67
```

The second variable was here assigned the ASCII code 32, which is the code of the blank space.

RUN 2:

```
Enter four characters: ABCDEFG
Your inputs are assigned to the variables as follows:
C1= A
C2= B
C3= C
C4= D
The corresponding ASCII codes are:
65 66 67 68
```

In the second case, the first four characters are read and the rest are ignored. If there were a subsequent READ statement in the program, it would start at the letter "E."

The End-Of-Line mark is treated like any other nonnumeric character. For example, if you test the program using these inputs:

```
AB   <Enter>
CD   <Enter>
```

the program will terminate after entering the first two characters and you will get an output like this:

RUN 3:

```
C1= A
C2= B
C3=        { CR }
C4=        { LF }

The corresponding ASCII codes are:
65 66 13 10
```

The third and the fourth characters contain CR and LF respectively, because when you press Enter, you send two characters to the INPUT file, CR and LF. Notice that the CR appears as a blank space (actually, it returns the cursor to the beginning of the line), while the LF advances to a new line.

The same thing will happen if you use two separate READ statements. To see this, replace the READ statement in the program by two READ statements:

```
READ(C1, C2);
READ(C3, C4);
```

When you run the program now, you will notice that if you type the first two characters and press Enter, the program will be terminated and you get the same output as in RUN 3.

Also, if you enter more characters than are required, only the first four will be read.

USING READLN FOR CHARACTER INPUT If you would like to enter your characters like this:

```
AB   <Enter>
CD   <Enter>
```

you have to get rid of the extra characters remaining in the file (the CR and the LF) by using the READLN statement.

In the following program two READLN statements are used, so you are able to enter two characters (or more) followed by Enter and start the next read with a clean buffer.

```
{ ------------------------------ figure 6-4 ------------------------------ }
PROGRAM CharReadln3(INPUT,OUTPUT);
CONST
   LF = CHR(10);
   CR = CHR(13);
VAR
   C1, C2, C3, C4 :CHAR;
BEGIN
   WRITE('Enter two characters: ');
   READLN(C1, C2);
   WRITE('Enter two characters: ');
   READLN(C3, C4);
   WRITELN('Your inputs have been assigned to the variables as follows:', CR, LF,
           'C1= ', C1, CR, LF,
           'C2= ', C2, CR, LF,
```

```
            'C3= ', C3, CR, LF,
            'C4= ', C4);
   WRITELN('The corresponding ASCII codes are:', CR, LF,
            ORD(C1),' ', ORD(C2),' ', ORD(C3),' ',ORD(C4))
END.
```

Sample run:

```
Enter two characters: abcd      <Enter>
Enter two characters: efgh      <Enter>
Your inputs have been assigned to the variables as follows:
C1= a
C2= b
C3= e
C4= f
The corresponding ASCII codes are:
97 98 101 102
```

INPUT OF MIXED TYPES It is legal to use one READ (or READLN) statement for mixed numeric and character data, but this requires extra attention. It is better to use a separate READLN statement for each type, as in the following program. This way is less prone to data entry errors.

```
{ ----------------------------- figure 6-5 ----------------------------- }
PROGRAM CharNumRead(INPUT,OUTPUT);
CONST
   LF = CHR(10);
   CR = CHR(13);
VAR
   A, B        :CHAR;
   X, Y        :INTEGER;
BEGIN
   WRITE('Enter two characters: ');
   READLN(A, B);
   WRITE('Enter two integers: ');
   READLN(X, Y);
   WRITELN('Your inputs have been assigned to the variables as follows:', CR, LF,
          'A= ', A, CR, LF,
          'B= ', B, CR, LF,
          'X= ', X, CR, LF,
          'Y= ', Y)
END.
```

RUN 1:

```
Enter two characters: ABCD
Enter two integers: 3 4
Your inputs have been assigned to the variables as follows:
A= A
B= B
X= 3
Y= 4
```

As you can see in the output, the extra characters (C and D) were skipped after the first READLN. Remember, however, that the rules of character entry still apply; in other words, if you press Enter after the first letter, a CR will be assigned to the variable "B." Here is the sample run:

RUN 2:

```
Enter two characters: A        <Enter>
B                              <Enter>
Enter two integers: 5 6
Your inputs have been assigned to the variables as follows:
A= A
B=            { B is assigned a CR }
X= 5
Y= 6
```

EXAMPLE: SCRAMBLING LETTERS The following example is good practice both for handling characters and building loops. The program asks you to enter four characters, then it displays all of the possible combinations of those characters. If you are a BASIC programmer, you would have had to use a lot of GOTOs to achieve these results. In Pascal the program is better structured.

```
{ ----------------------------- figure 6-6 ---------------------------- }
PROGRAM Scrambling(INPUT,OUTPUT);
TYPE
   ScrambleArray = Array[1..4] OF CHAR;
VAR
   A              :ScrambleArray;
   I1, I2, I3, I4 :INTEGER;
BEGIN
   WRITE('Enter four letters: ');
   READ(A[1], A[2], A[3], A[4]);
   FOR I1 := 1 TO 4 DO
      BEGIN
```

```
        FOR I2 := 1 TO 4 DO
          BEGIN
            IF I2 <> I1 THEN
              FOR I3 := 1 TO 4 DO
                BEGIN
                  IF I3 <> I1 THEN
                    IF I3 <> I2 THEN
                      BEGIN
                        I4 := 10 - (I1 + I2 + I3);
                        WRITELN(A[I1],' ',A[I2],' ',
                                A[I3],' ',A[I4]);
                      END     { End of IF }
                END           { End of I3 loop }
          END     { End of I2 loop }
      END     { End of I1 loop }
END.
```

Sample run:

```
Enter four letters: ABCD
A B C D
A B D C
A C B D
A C D B
A D B C
A D C B
B A C D
B A D C
B C A D
B C D A
B D A C
B D C A
C A B D
C A D B
C B A D
C B D A
C D A B
C D B A
D A B C
D A C B
D B A C
D B C A
D C A B
D C B A
```

An array "A" of four elements (of the type CHAR) is used to hold the four characters, and three nested loops are used to build the different combinations of the elements. The algorithm is based on choosing four different indexes corresponding to the four different array elements.

Note that all of the BEGIN-END blocks (except the innermost one) are optional and are used only for clarity.

6-4 READING A LINE OF TEXT: EOLN

The EOLN function is a boolean function used to detect the end of the line during reading of the INPUT file. The function is FALSE until the End-Of-Line mark is detected, then it becomes TRUE.

This function is useful when you do not know the number of characters to expect.

In order to read a line of text up to (but not including) the End-Of-Line mark, you can use a loop like this:

```
WHILE NOT EOLN DO
   BEGIN
      READ(Ch);
      ...
   END;
```

The READ statement will continue to read characters until the End-Of-Line mark is detected, thus terminating the WHILE loop. Notice, however, that the End-Of-Line mark is still in the buffer and could be read by any subsequent READ statement, so before any subsequent read you have to clean the buffer with a READLN.

EXAMPLE: CHARACTER COUNTER The following program reads a line of text from the keyboard and displays the number of characters in the line. The program will continue to read the characters you type until you press Enter, at which time it displays the result.

```
{ ---------------------------- figure 6-7 -------------------------- }
PROGRAM CharCounter1(INPUT,OUTPUT);
VAR
   Ch      :CHAR;
   Counter :INTEGER;
BEGIN
```

```
    Counter := 0;
    WHILE NOT EOLN DO
      BEGIN
        READ(Ch);
        Counter := Counter + 1
      END;
    WRITELN;
    WRITELN('Number of characters= ', Counter)
END.
```

DRILL 6-3

Modify the previous program to count only the alphabetic characters in the text.

6-5 READING A FILE OF TEXT: EOF

Another boolean function EOF is used to detect the *End-Of-File* mark. The function is FALSE until the End-Of-File mark is reached, at which time it becomes TRUE. When using the keyboard for input, the end of file is reached if you press Ctrl-Z (ASCII 26). This function is useful for reading several lines of text (a file). You can use EOF along with EOLN to read and analyze several lines of text as follows:

```
WHILE NOT EOF DO
  BEGIN
    WHILE NOT EOLN DO
      BEGIN
        READ(Ch);
        ...   { Processing data }
      END;    { End of line }
    READLN    { Advance the pointer }
  END;        { End of file }
```

In this code, the file is read line by line. After a complete line has been read, the EOLN function becomes TRUE and no more characters are read from this line. The READLN statement is then used to advance the pointer to the beginning of the next line. The program ends when the End-Of-File mark is detected and the outer loop is terminated. Let us see an example.

EXAMPLE 1: FREQUENCY COUNTER The following program asks you to enter a letter. Then it starts reading whatever you type from the keyboard. When you press Ctrl-Z the program ends and displays how many times the specified letter was repeated in the file.

```
{ ------------------------- figure 6-8 ------------------------- }
PROGRAM FreqCounter1(INPUT,OUTPUT);
VAR
   Ch, SpecificChar        :CHAR;
   Counter, FreqCounter    :INTEGER;
BEGIN
   Counter := 0;
   FreqCounter := 0;
   WRITE("Enter the required letter: ");
   READLN(SpecificChar);
   WRITELN("Start typing. Press Ctrl-Z to finish.");
   WHILE NOT EOF DO
      BEGIN
         WHILE NOT EOLN DO
            BEGIN
               READ(Ch);
               IF (Ch >= "A") AND (Ch <= "Z") OR
                     (Ch >= "a") AND (Ch <= "z") THEN
                  Counter := Counter + 1;
               IF Ch = SpecificChar THEN
                  FreqCounter := FreqCounter + 1;
            END;
         READLN
      END;
   WRITELN("Total number of letters= ", Counter);
   WRITELN("The letter "", SpecificChar, "" was repeated ",
           FreqCounter, " time(s)");
   WRITELN("Frequency of repetition= ", FreqCounter/Counter*100:2:2, "%")
END.
```

The specific letter is assigned to the variable "SpecificChar" and compared to the input letter "Ch." If the comparison is TRUE, the "FreqCounter" is incremented by one. The total number of letters is accumulated in the variable "Counter." The frequency of repetition of the letter is calculated by dividing "FreqCounter" by "Counter" and multiplying the result by 100.

Sample run:

```
Enter the required character: a
Start typing. Press Ctrl-Z to finish.
This is a test to count the repetition frequency
of the letter "a" in a keyboard file
^Z

Total number of letters= 67
The letter 'a' was repeated 4 time(s)
Frequency of repetition= 5.97%
```

6-6 STRING MANIPULATION

In chapter two you learned how to declare, read, and write variables of the type STRING, which was introduced by the modern Pascal implementations (such as Turbo, UCSD, and Macintosh). You also learned how to use the function LENGTH to count the number of letters in a string. In this section you are introduced to more string features that help in manipulating text.

TIPS ON STRING INPUT/OUTPUT For both input and output, you may either treat a string variable as one unit, or you may treat it as an array whose elements are the characters that make up the string. Look at this simple program, which reads a string variable and displays it character by character, with each character on a separate line (using the LF character).

```
{ ------------------------------ figure 6-9 ------------------------------ }
PROGRAM String1(INPUT,OUTPUT);
CONST
  LF = CHR(10);
VAR
  Name :STRING[30];
  I    :INTEGER;
BEGIN
  WRITE('Please enter a name: ');
  READLN(Name);
  FOR I := 1 TO LENGTH(Name) DO
    WRITE(Name[I],LF)
END.
```

Sample run:

```
Please enter a name: PASCAL
P
 A
  S
   C
    A
     L
```

EXAMPLE: SORTING NAMES You may build an array of the type STRING to store related items such as names or addresses. In this way, you can sort names in alphabetical order using the same algorithm which you have used before to sort numbers. Each two strings are compared character by character. So, the following expressions are TRUE:

```
'Able' < 'Baker'
'Baker' < 'Charlie'
'Charley' < 'Charlie'
```

All uppercase letters are greater than lowercase letters. Also, the leading and trailing spaces are included in the comparison. The ASCII code of the blank space (32) is less than that of any letter or digit. In the following program an array of four names is read, sorted, and displayed.

```
{ ------------------------------ figure 6-10 ------------------------------ }
PROGRAM SortStrings(INPUT,OUTPUT);
CONST
   Tab = '          ';
   NumOfElements = 4;
TYPE
   StringArray = ARRAY[1..NumOfElements] OF STRING[30];
VAR
   Name    :StringArray;
   I, J    :INTEGER;
   Temp    :STRING[30];
BEGIN
{ Read the array elements }
{ ----------------------- }
   FOR I := 1 TO NumOfElements DO
      BEGIN
         WRITE('Please enter name #', I, ': ');
         READLN(Name[I])
      END;
```

```
{ Sort names }
{ ---------- }
   FOR I := 1 TO NumOfElements-1 DO
      FOR J := I+1 TO NumOfElements DO
         IF Name[I] > Name[J] THEN
            BEGIN
               Temp := Name[I];
               Name[I] := Name[J];
               Name[J] := Temp
            END;
      { End of inner and outer loops }
{ Display sorted names }
{ ------------------- }
   WRITELN('Serial #    Name');
   WRITELN('-----------------------------');
   FOR I := 1 TO NumOfElements DO
      WRITELN(I:2, Tab, Name[I])
END.
```

Sample run:

```
Please enter name #1: Rigby, Peter
Please enter name #2: Berlin, Amy
Please enter name #3: Sanders, Dale
Please enter name #4: Brady, Clark

Serial #    Name
-----------------------------
 1          Berlin, Amy
 2          Brady, Clark
 3          Rigby, Peter
 4          Sanders, Dale
```

DRILL 6-4

Write a program to scramble three strings. The following is an example of the output for the strings "WHO," "ARE," and "YOU":

```
WHO ARE YOU
WHO YOU ARE
ARE WHO YOU
ARE YOU WHO
YOU WHO ARE
YOU ARE WHO
```

6-7 STRING FUNCTIONS AND PROCEDURES

When working with text editors, you sometimes need to cut and paste, delete a part from here, and insert a part there. The tools that make these operations possible are included in the modern implementations of Pascal to help the programmer process strings. Some of them are called functions because they return a value which replaces the function call (e.g., LENGTH). Others are called *procedures*, as they perform specific operations that do not necessarily return a value (e.g., WRITELN). They are all shown in table 6-1.

In addition to these tools you may find more functions, procedures, or operators in a specific implementation, but here we are concerned only with the most common tools, which are almost standardized.

Table 6-1 String functions and procedures

Form	*Use*
Functions:	
LENGTH(str)	Returns the number of character in the string "str."
CONCAT(str1, str2,...)	Returns the string formed by concatenating "str1," "str2,"...
COPY(str, pos, len)	Returns a substring from the string "str," starting at the position "pos," with length "len."
POS(str1, str2)	Returns the position of the first occurrence of the first character of "str1" within "str2." If "str1" does not occur within "str2" it returns zero.
Procedures:	
INSERT(str1, str2, pos)	Inserts the string "str1" into the string "str2," at the position "pos."
DELETE(str, pos, len)	Deletes a substring from a string "str" starting from position "pos" with length "len."

CONCAT As an example of using the function CONCAT, you can concatenate the three strings 'John ', 'M. ', and 'Smith' and assign the result to a string variable "Name," as follows:

```
Name := CONCAT('John ','M. ','Smith');
```

Now the variable "Name" contains the complete name: "John M. Smith." In Turbo Pascal the operator "+" may also be used to concatenate strings:

```
Name := 'John '+ 'M. '+ 'Smith';
```

The variable "Name" has the same contents as before.

COPY Using the function COPY you can do the opposite, i.e. extract a substring from the string "Name." The following statement extracts the first name from the string "Name" and assigns it to the variable "FirstName":

```
FirstName := COPY(Name, 1, 4);
```

As you can see, you have to include the starting position of the extracted substring (1 in this case) and the length of the substring (4 in this case).

POS The function POS returns an integer that indicates the position of the first occurrence of a substring in a string. For example, the statements:

```
Str1 := 'This is a test';
WRITELN(POS('is', Str1));
```

result in displaying the number "3," which is the position of the letter "i" in "This."

DELETE To delete the substring 'Smith' from a name string, use the DELETE procedure as follows:

```
DELETE(Name, 9, 5);
```

Note that the substring 'Smith' starts at the 9th position and contains 5 characters. Using a procedure changes the value of the original variable "Name," while using a function does not. If you checked the contents of the variable now, it would be 'John M. '.

INSERT Don't worry, because you can insert the substring in the right place again. Use the INSERT procedure to put the last name "Smith" back in the string:

```
INSERT('Smith', Name, 9)
```

Now the variable "name" contains 'John M. Smith'.

The following program demonstrates the use of string functions. It accepts from you the first, middle, and last name and produces the complete name, including the trailing spaces. Also, the middle name is converted to an initial.

```
{ ---------------------------- figure 6-11 ----------------------------- }
PROGRAM StringFunctions1(INPUT,OUTPUT);
VAR
  Name                :STRING[30];
  First, Middle, Last :STRING[10];
BEGIN
```

```
   WRITE('Please enter your first name: ');
   READLN(First);
   First := CONCAT(First, ' ');
   WRITE('Please enter your middle name: ');
   READLN(Middle);
   Middle := COPY(Middle, 1, 1);
   Middle := CONCAT(Middle, '. ');
   WRITE('Please enter your last name: ');
   READLN(Last);
   Name := CONCAT(First, Middle, Last);
   WRITELN;
   WRITELN('Your complete name is: ',Name)
END.
```

Sample run:

```
Please enter your first name: Sally
Please enter your middle name: Ann
Please enter your last name: Abolrous

Your complete name is: Sally A. Abolrous
```

DRILL 6-5

Modify the last program to make it capitalize the first letter of each name if lowercase.

SUMMARY

In this chapter you learned how the input statements **READ** and **READLN** work with numeric values and characters. You also learned how to use the End-Of-Line function EOLN to read a line of text from the keyboard, and the End-Of-File function EOF to read a file of text. You also learned some of the important string-processing functions and procedures which are available in the modern implementations of Pascal:

CONCAT
COPY
POS
INSERT
DELETE

Most importantly, through the examples and drills you gained experience in text processing with both strings and individual characters.

DAY THREE

CHAPTER SEVEN

PROGRAM ARCHITECTURE

7-1 PROGRAMS AND SUBPROGRAMS

When you deal with real applications, the problems get more complex than those you have met so far, so you usually have to break the main problem down into simpler subproblems and program each individually in a *subprogram*. The subprograms are then combined together to build up the complete program. If you can break your application down into the smallest possible modules, you will find that many of them are common problems such as sorting names or numbers. This means that you can write some generic subprograms and use them later in different applications. Another advantage of using subprograms is that you can thus avoid the repetition of several statements to print a header or display a menu; you can program such tasks as subprograms and call them whenever needed. In Pascal you can divide your program into smaller subprograms called *procedures* and *functions*. Actually, the Pascal language itself is made up from predefined procedures and functions. When the compiler encounters a WRITELN statement in a program, for example, the predefined procedure WRITELN is invoked to perform the required task.

7-2 PROCEDURES

If divided into procedures, the main body of the "Scores" program from chapter five might look something like this:

```
BEGIN
   ReadScores;
   GetAverage;
   DisplayResults
END.
```

The main program contains only three *calls*, each of them the name of a procedure which performs a specific task. The procedure "ReadScores" reads the array of the scores, "GetAverage" calculates the average score, and "DisplayResults" displays the results. As you can see, a user-defined procedure is called by its name just like any standard procedure.

PROCEDURE DEFINITION Before calling a procedure it must be defined in the *subprogram section*, which is the last section of the declaration part. The following is a complete list of all the sections of the declaration part:

- **LABEL** section
- **CONST** section
- **TYPE** section
- **VAR** section
- **PROCEDUREs** and **FUNCTIONs** section

A procedure definition is very similar to a program definition in that it consists of a header, a declaration part, and statements. Let us begin with a simple procedure to draw a line 20 characters long.

```
{ ------------------------------ figure 7-1 ------------------------------ }
PROGRAM Procedures1(OUTPUT);
{ ----------- Beginning of Procedure ----------- }
PROCEDURE DrawLine;
CONST
   Dash = '-';
   LineLength = 20;
VAR
   Counter :INTEGER;
BEGIN
   FOR Counter := 1 TO LineLength DO
      WRITE(Dash);
   WRITELN
END;
{ -------------- End of Procedure -------------- }
{ ---------------- Main program ---------------- }
BEGIN
   WRITELN;
   DrawLine;
   WRITELN('** THIS IS A TEST **');
   Drawline
END.
```

The output is:

```
--------------------
** THIS IS A TEST **
--------------------
```

There are no variables or constants in the main program here, so the declaration part contains only the procedure definition. The definition starts with the procedure header:

```
PROCEDURE Drawline;
```

The header includes the name of the procedure (DrawLine), which must be a valid identifier. Then comes the declaration part:

```
CONST
   Dash = '-';
   LineLength = 20;
VAR
   Counter :INTEGER;
```

The declaration part of the procedure includes the same sections as that of the main program. In our example, two named constants and a variable were declared. Then come the statements of the procedure which represent the task to be done (drawing a line), enclosed in a block.

```
BEGIN
   FOR Counter := 1 TO LineLength DO
      WRITE(Dash);
   WRITELN
END;
```

Notice that the END statement in a subprogram is terminated by a semicolon rather than a period.

In the main program, the procedure is called twice to draw a line both before and after the displayed text.

PASSING VALUES TO PROCEDURES The procedure "DrawLine" is used to draw a line of a specific length (20), which may not be useful for any other application. In the following program the procedure is modified to draw a line whose length varies according to the length of the displayed text. When you run the program it asks you to enter a sentence, then displays the sentence between two lines of the same length as that sentence. Try the program first and then read the discussion.

```
{ --------------------------- figure 7-2 ----------------------------- }
PROGRAM Procedures2(OUTPUT);
```

```
VAR
   Len           :INTEGER;
   TestSentence  :STRING;
{ ------------ Beginning of Procedure ------------ }
PROCEDURE DrawLine(LineLength :INTEGER);
CONST
   Dash = '-';
VAR
   Counter :INTEGER;
BEGIN
   FOR Counter := 1 TO LineLength DO
      WRITE(Dash);
   WRITELN
END;
{ --------------- End of Procedure --------------- }
{ ----------------- Main program ----------------- }
BEGIN
   WRITE('Please enter a sentence: ');
   READLN(TestSentence);
   Len := LENGTH(TestSentence);
   WRITELN;
   DrawLine(Len);
   WRITELN(TestSentence);
   Drawline(Len)
END.
```

Sample run:

```
Please enter a sentence: Learn Pascal in Three Days
--------------------------
Learn Pascal in Three Days
--------------------------
```

Instead of defining the number of dashes as a constant, the length of the sentence is declared in the main program as a variable "Len." After the sentence is entered, its length is calculated and passed to the procedure as a *parameter*. The procedure call in this case becomes:

```
DrawLine(Len);
```

The procedure header must also include a receiver parameter:

```
PROCEDURE DrawLine(LineLength :INTEGER);
```

Between the parentheses comes the parameter "LineLength," followed by a colon, followed by the type of the parameter (INTEGER).

When the procedure is invoked, the value of the variable "len" (from the main program) is passed to the procedure and assigned to the variable "LineLength," where it is used in processing. The variable "len" is called the *actual parameter*, and the variable "LineLength" is called the *formal parameter*. After the procedure has been executed, the control is transferred back to the main program, and execution resumes at the next statement following the procedure call. Except during the procedure execution, the value of the formal parameter is undefined.

You may use literal values as actual parameters to call the procedure, such as:

```
DrawLine(30);
```

This call results in the drawing of a line 30 characters long.

(Note: Functions and procedures can also be passed as parameters, but many implementations forbid this).

When a value is used as a parameter, it is said that the parameter is passed *by value*; if the parameter is a variable, it is said to be passed *by reference*.

A procedure call may contain more than one parameter, like this:

```
Process(A, B, C);
```

The number of actual parameters in the procedure call must be the same as the number of formal parameters, which means that the procedure header may look something like this:

```
PROCEDURE Process(X, Y :INTEGER; Z :REAL);
```

The variables "A" and "B" in the calling program must be of the type INTEGER as they correspond to "X" and "Y" respectively, while the variable "C" must be of the type REAL as it corresponds to "Z." Note the semicolon that separates the declarations in the procedure header.

In brief, the actual and formal parameters must match in number, type, and position.

DRILL 7-1

Modify the last program so that you can pass to the procedure the type of line character ('-' or '*' etc..), and have the output displayed in the middle of the line (assume that the line is 80 characters wide). This is a sample run of the required program:

```
Please enter a sentence: Learn C in Three Days
Please enter the line character: *
                              *********************
                              Learn C in Three Days
                              *********************
```

PASSING BACK VALUES FROM PROCEDURES A procedure may be used to change the value of a variable and pass it back to the calling program. In such a case, the formal parameters must be preceded by the word VAR. Consider the case of a procedure that receives the value of two variables and returns the cube of each. The procedure header might look something like this:

```
PROCEDURE CubeThem(VAR X, Y :REAL);
```

You can only pass parameters to this procedure by reference:

```
CubeThem(A, B);
```

The values of "A" and "B" will be passed to the procedure, substituted for "X" and "Y" respectively, cubed, and sent back to the calling program. It is illegal in this case to use literal values or expressions as actual parameters.

When formal parameters are preceded by the word VAR they are called *variable parameters*; otherwise they are *value parameters*.

The general form of the procedure header is:

> **PROCEDURE** name;
> or
> **PROCEDURE** procedure-name(formal-parameter-list);

The general form of a procedure call is:

> procedure-name;
> or
> procedure-name(actual-parameter-list);

The following program is an example of using both types of formal parameters. It demonstrates the same logic as the "PowerOperator" program (figure 2-2)

does but uses a procedure to receive the base and the power and send back the result.

```
{ ----------------------------- figure 7-3 -------------------------------}
PROGRAM VarParms(INPUT,OUTPUT);
VAR
   a, b, c :REAL;
{ ----------- Procedure Definition -------------- }
PROCEDURE PowerOperator(X, Y :REAL; VAR Z:REAL);
BEGIN
     Z := EXP(LN(X)*Y)
END;
{ ---------------- Main Program ----------------- }
BEGIN
   WRITE('Enter the base and the exponent separated by a space:');
   READLN(a,b);
   PowerOperator(a, b, c);
   WRITELN('The value of ',a:0:2,' raised to the power of ',b:0:2,' is ',c:0:2)
END.
```

Sample run:

```
Enter the base and the exponent separated by a space:2 5
The value of 2.00 raised to the power of 5.00 is 32.00
```

Notice in the procedure that X and Y were declared as value parameters because they only receive values from the calling program, while Z was declared as a variable parameter because it sends back the result.

7-3 GLOBAL AND LOCAL VARIABLES

Both the formal parameters and the variables declared in a procedure are called *local variables* because they are accessible only within their procedure; in other words, they are invisible to the main program or to any other subprogram. The variables declared in the main program, on the other hand, are called *global variables* because they are accessible from any program unit. In the program 7-2, for example, the variable "TestSentence" is a global variable and may be accessed from the procedure "DrawLine" without passing it as a parameter. Any assignment to this variable in the procedure will change its value in the main program. The local variable "Counter," however, is not accessible from the main program.

Consider now the case if you declared two variables with the same name (such as X), one in the main program and one in a procedure. The redeclaration of the global variable "X" in a procedure will create a local variable with the same name and hide the global variable from the procedure. This means you will have two different variables that correspond to two different locations in memory. When the procedure exits, there will be one global variable with the name "X." These restrictions help the programmer not to modify the value of a global variable from a subprogram by accident.

The variables in the main program can only be modified from other procedures if they are global (and not redeclared in the procedure) or are passed by reference as variable parameters to the procedure. Accessing global variables from a subprogram is not recommended, as it repeals the modularity of the program. Using parameters is safer, and it also keeps the subprogram independent and useful with different programs.

EXAMPLE: SORTING PROCEDURE Go back to program 5-5, and let us split it into generic procedures. This program was used to read, sort, and display an array of six elements. What you need to do now is write three procedures that read, sort, and display an array of any size. By passing the array and the number of elements to the procedures, the same results will be achieved as before. The main body of the program will contain only three calls:

```
ReadNumbers(ArraySize, Numbers);
SortNumbers(ArraySize, Numbers);
PrintNumbers(ArraySize, Numbers);
```

In this way, any one of the three procedures can be used in any program. One important point to mention here is that when you pass an array to a procedure or function, it must be declared in the TYPE section. The formal parameters in the procedure header will then look something like this:

```
PROCEDURE ReadNumbers(L: INTEGER; VAR R :NumbersArray);
```

The parameter "L" corresponds to "ArraySize," and the array "R" corresponds to the array "Numbers." As you can see in the parameter declaration it is of the type "NumbersArray," which is the same type as the array "Numbers." Here is the complete program:

```
{ ----------------------------- figure 7-4 ----------------------------- }
PROGRAM Sorting(INPUT,OUTPUT);
CONST
  ArraySize = 6;
TYPE
```

```
   Range         = 1..ArraySize;
   NumbersArray = ARRAY[Range] OF INTEGER;
 VAR
   Numbers :NumbersArray;
 { ----------------- Read procedure --------------- }
 PROCEDURE ReadNumbers(L: INTEGER; VAR R :NumbersArray);
 VAR
   I :INTEGER;
 BEGIN
   FOR I := 1 TO L DO
      BEGIN
         WRITE('Enter element #', I,': ');
         READLN(R[I])
      END
 END;
 { ----------------- Sort procedure --------------- }
 PROCEDURE SortNumbers(M: INTEGER; VAR S :NumbersArray);
 VAR
   I, J, Pot :INTEGER;
 BEGIN
   FOR I := 1 TO M-1 DO
      FOR J := I+1 TO M DO
         IF S[I] > S[J] THEN
            BEGIN                                { swap contents }
               Pot := S[J];
               S[J] := S[I];
               S[I] := Pot
            END
 END;
 { ---------------- Print procedure --------------- }
 PROCEDURE PrintNumbers(N: INTEGER; T :NumbersArray);
 VAR
   I :INTEGER;
 BEGIN
   WRITELN;
   WRITE('The sorted array is: ');
   FOR I := 1 TO N DO
      WRITE(T[I],'  ');
   WRITELN;
 END;
 { -------------- Main Program ------------------ }
 BEGIN
   ReadNumbers(ArraySize, Numbers);
   SortNumbers(ArraySize, Numbers);
```

```
   PrintNumbers(ArraySize, Numbers);
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

Sample run:

```
Enter element #1: 44
Enter element #2: 22
Enter element #3: 8
Enter element #4: 1
Enter element #5: 667
Enter element #6: 3
The sorted array is: 1  3  8  22  44  667
Press ENTER to continue..
```

Note that the array is passed as a variable parameter to the procedures which are expected to change the value of the array (e.g., "ReadNumbers" and "SortNumbers"), but there was no need to do that for the procedure "PrintNumbers," which displays the array without returning any value to the main program. In the latter case the array was passed as a value parameter. Notice also the use of local variables in different procedures, which makes each an independent unit. If any of these procedures has to be used with a different type of array, you need only change the type "NumbersArray" or use the same type name for the new array in the main program. In this example it is possible to use procedures without any parameters at all and process the global variables directly, but in that case you would have to use the same variable names in all of the procedures and the main program, which is a lot of effort and also entails the risk of dealing with global variables.

TIP Like arrays, enumerated types and subranges must be declared in the TYPE section if they are to be used as formal parameters in a subprogram call.

7-4 FUNCTIONS

A function is a subprogram that returns a value, which is then assigned to the function name in the calling program. Like predefined functions, user-defined functions have one or more parameters. The function definition comes in the subprogram section of the declaration part and includes a header, a declaration part, and statements. Look at this header of a function that returns the average of three numbers:

```
FUNCTION Avg(X, Y, Z :REAL) :REAL;
```

The header is similar to the procedure header except that the type of the return value follows the function header (:REAL). You can call this function using statements like these:

```
D := Avg(A, B, C);
WRITELN(Avg(F, G, H):2:2);
WRITELN(Avg(94, 33.5, 45*1.2):2:2);
```

As you can see, the parameter may be a literal constant, an expression, or a variable.

The function header takes the following form:

FUNCTION function-name(formal-parameter-list) :return-type;

In this program the function "Avg" is demonstrated.

```
{ ------------------------------ figure 7-5 ------------------------------ }
PROGRAM Functions1(INPUT, OUTPUT);
VAR
  A, B, C :REAL;
{ ----------- Beginning of Function ------------ }
FUNCTION Avg(X, Y, Z :REAL) :REAL;
BEGIN
  AVG := (X + Y + Z) / 3
END;
{ -------------- End of Function --------------- }
{ ---------------- Main program ---------------- }
BEGIN
  WRITE('Enter three numbers: ');
  READLN(A, B, C);
  WRITELN('The average is= ', Avg(A, B, C):0:2)
END.
```

Sample run:

```
Enter three numbers: 2 3 8
The average is= 4.33
```

Like procedures, functions are independent subprograms. All parameters, variables, and constants declared within the function body are local to it and are invisible to other program units. In a function subprogram, the function must be assigned a value.

TIP In a function subprogram, the function name cannot be treated like a variable; i.e., it may not be involved in expressions. It may only be assigned a value.

DRILL 7-2

Write a function to return the maximum number in a one-dimensional array and include the function in a program. You may use any procedures you wrote before to build the program.

7-5 TIPS ON THE SCOPE OF VARIABLES

The following program frame consists of three program units, procedure "Kid1," procedure "Kid2," and the main program "Parent." According to the rules of variable scope, any variable declared in "Parent" (global variable) is accessible to both "Kid1" and "Kid2" unless it is redeclared locally in either of them. On the other hand, any local variable declared in "Kid1" is hidden from both "Parent" and "Kid2." The same thing applies for "Kid2" variables. If you consider the main program as a parent and the subprograms as kids, it then follows that whatever belongs to the parent belongs to the kids, but the opposite is not valid. In other words, the kids inherit everything from the parent, but each one of them has his own property, which is not inherited by a parent or a sibling.

```
{ ------------------------------ figure 7-6 -------------------------- }
PROGRAM Parent;
    { -------------- PROCEDURE KID1 -------------- }
    PROCEDURE Kid1(...);
    ...
    BEGIN
    ...
    END; { --------- END OF PROCEDURE KID1 --------- }
    { --------------- PROCEDURE KID2 ------------- }
    PROCEDURE kid2(...);
    ...
    BEGIN
    ...
    END; { --------- END OF PROCEDURE KID2 --------- }
{ ------------------------------ MAIN PROGRAM --------------------------- }
BEGIN
...
END.
```

Either of the two procedures may be called from the main program. The procedure "Kid1" may also be called from "Kid2" because it has already been defined, but the procedure "Kid2" cannot be called from "Kid1" because it has not yet been defined. There is a way to get around this restriction using a *forward declaration* by including the header of "Kid2," followed by the keyword FORWARD, at the beginning of the program, like this:

```
PROGRAM Parent;
{ Forward declaration of Kid2 }
    PROCEDURE Kid2(...); FORWARD;
{ Definition of Kid1 }
    PROCEDURE Kid1(...);
        ...
{ Definition of Kid2 }
    PROCEDURE kid2(...);
        ...
{ Main program }
        ...
```

Now take a look at the new program structure in the following figure. The procedure "GrandKid" is defined inside the procedure "Kid," which means that "Kid" has become the parent of another subprogram. In such a case, any variable in "Kid" is global in "GrandKid," and so are the variables of the "Parent" (unless any of them is redeclared in "GrandKid"). The local variables in "GrandKid," however, are not accessible to either "Kid" or "Parent."

```
{ ------------------------------ figure 7-7 --------------------------- }
PROGRAM Parent;
    { ------------------ PROCEDURE KID ------------------ }
    PROCEDURE Kid(...);
    ...
        { ----------- PROCEDURE GRANDKID ----------- }
        PROCEDURE GrandKid(...);
        BEGIN
        ...
        END; { ----- END OF PROCEDURE GRANDKID ----- }
    BEGIN
    ...
    END; { ------------- END OF PROCEDURE KID ------------- }
{ ---------------------------- MAIN PROGRAM ---------------------------- }
BEGIN
...
END.
```

To summarize:

1. The scope of a variable is the program unit in which it is declared.
2. A global variable is accessible in any program unit unless it is redeclared locally in that unit.
3. A local variable is not accessible outside the program unit in which it is declared. It is, however, accessible to any subprogram defined within this program unit unless redeclared inside that sub-subprogram.
4. Any subprogram can be called from any program unit as long as its definition (or its forward declaration) preceded the call.

7-6 RECURSION

A function or procedure may call itself, a property called recursion. The factorial function is a good example of recursion. You know (from chapter four) that the factorial of a number "X" can be obtained from the relation:

```
factorial(X) = X * factorial(X-1)
```

In other words, to get the factorial of 4 you multiply 4 by the factorial of 3; to get the factorial of 3 you multiply 3 by the factorial of 2, etc. This continues until you reach the value 1.

Here is the program that contains the factorial function.

```
{ ------------------------------ figure 7-8 ------------------------------ }
PROGRAM FunctionRecursion(INPUT, OUTPUT);
VAR
  A :INTEGER;
{ ------------- Function Definition -------------- }
FUNCTION Factorial(X :INTEGER) :REAL;
BEGIN
  IF X <= 1 THEN
    Factorial := 1
  ELSE
    Factorial := X * Factorial(X-1);
END;
{ --------------- End of Function ---------------- }
{ ----------------- Main program ----------------- }
BEGIN
  WRITE('Enter a number: ');
```

```
   READLN(A);
   WRITELN('The Factorial of ', A,' = ', Factorial(A):0:0)
END.
```

Sample run:

```
Enter a number: 6
The Factorial of 6 = 720
```

Notice in the function program that in the statement:

```
    Factorial := X * Factorial(X-1);
```

the left side contains the name of the function, while in the right side there is a call of the function to calculate the factorial of X-1. This process will continue until the condition terminates the function.

DRILL 7-3

Write the factorial subprogram as a procedure and compare it to the factorial function.

SUMMARY

In this chapter you learned about the Pascal program structure. You know how to divide your program into subprograms, whether functions or procedures. These are important points to remember:

1. A subprogram is declared in the last section of the declaration part and consists of a header, a declaration part, and statements.

   The header of a procedure takes the form:

   PROCEDURE name;

   or

   PROCEDURE procedure-name(formal-parameter-list);

   The header of a function takes the form:

   FUNCTION function-name(formal-parameter-list) :return-type;

2. A procedure is called by its name exactly like a statement. When parameters are used in a procedure call, they must match the parameters in the procedure header. Procedure parameters are either value or variable parameters. A variable parameter is used when it is required to have the procedure change the value of the parameter.

3. A function is usually called as part of an expression; it returns a single value that replaces the name of the function in that expression.
4. You now know that each variable has a scope, and you learned the rules that control the scope and the relationship between global and local variables.

CHAPTER EIGHT

SETS AND RECORDS

8-1 SETS

If you would like to test a character to see if it is an uppercase letter, you may use the following condition:

```
READ(Character);
IF (Character >= 'A') AND (Character <= 'Z') THEN ...
```

There is a simpler way to express the same condition. Take a look at this statement:

```
IF Character IN ['A'..'Z']  THEN ...
```

This expression speaks for itself; it is almost plain English. It says: "if the character is IN the set of uppercase letters ['A'..'Z'] then ..."

In a similar way, you can express any set of items such as:

`['a'..'z']`	the set of lowercase letters,
`['A'..'Z', 'a'..'z']`	the set of all letters, and
`[0..9]`	the set of digits.

The set is a structured data type that may include unordered elements (or *members*). You can express a set constant by listing its elements between brackets separated by commas. Unlike arrays, the order of elements in a set is not important. For example, the set [1,3,5,7], which represents the set of odd numbers between one and seven, is the same as the set [1,7,5,3]. This indicates another difference between sets and arrays. In arrays you can access any element by its position in the array, but with sets you cannot access individual elements. You can only test a data item to see if it is a member of the set using the IN operator. If the elements of a set form a continuous subrange, you may use the two periods (..); for example, the set [1,2,3,4,6,8] can be written as

[1..4,6,8]. The elements of a set can be of any ordinal type, but all of the elements must be of the same type, which is called the *base type*.

8-2 SET DECLARATION AND ASSIGNMENT

You can declare a set variable using the keywords SET OF, as in this example where a set of the base type CHAR is declared:

```
VAR
   LowerCase :SET OF CHAR;
```

After this declaration you can assign the variable "LowerCase" a set constant of the base type CHAR, for example:

```
LowerCase := ['a'..'z'];
```

You may then test a variable of the type CHAR for *membership* in this set using an expression like:

```
IF Character IN LowerCase THEN ...
```

Note that if you use the expression "IN ['a'..'z']" there is no need for declarations.

As with other structured types, it is preferable to declare sets in the TYPE section; you can then use this type in the VAR section to declare variables. The declaration takes the form:

type-identifier = **SET OF** base-type;

Here is an example:

```
TYPE
   Days = (Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);
   Languages = (C,CPP,Pascal,Fortran,Basic,Cobol,Assembly);
   Digits = SET OF 0..9;
   Lowercase = SET OF 'a'..'z';
   Uppercase = SET OF 'A'..'Z';
   DaySet = SET OF Days;
   LanguageSet = SET OF Languages;
   CharacterSet = SET OF CHAR;
VAR
   WholeWeek, WorkingDays, WeekEnd :DaySet;
   OddNum, EvenNum, Numbers        :Digits;
   Small                           :Lowercase;
   Capital                         :Uppercase;
```

```
ProgCodes, HLL, LLL, MLL        :LanguageSet;
Alphabet                        :CharacterSet;
```

In these declarations, variables such as "Weekend," "WorkingDays," and "WholeWeek" are all sets of the base type "Days." Any of these set variables may be assigned one or more elements of the enumeration "Days," such as:

```
WeekEnd := [Saturday,Sunday];
WorkingDays := [Monday..Friday];
WholeWeek := [Monday..Sunday];
```

The last statement is equivalent to the statement:

```
WholeWeek := [Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday];
```

The value of a set variable is undefined until it is assigned a value. When you assign a set constant to a set variable, their base types must be *compatible*, i.e., they must be of the same type, subranges of the same type, or one of them must be a subrange of the other. Here are more assignments:

```
OddNum := [1,3,5,7,9];
EvenNum := [2,4,6,8];
ProgCodes := [C..Assembly];
LLL := [Assembly];
```

The *empty set* is a set with no members and is denoted by the constant [ ]. You may assign this constant to any set variable of any base type, for example:

```
OddNum := [];
```

RULES AND RESTRICTIONS The following are the main rules and restrictions that control the use of sets:

1. There is usually a limit on the maximum number of elements of a set. This limit varies with different Pascal implementations; in Turbo Pascal, for example, it is 255. The declaration "SET OF INTEGER" is not allowed because the range of integers exceeds this maximum number, but you can get around that by using subranges such as "SET OF 0..99."

   In some implementations, the declaration "SET OF CHAR" is not allowed either, in which case a subrange of the type CHAR might be used.

2. You may assign a set to another set:

   ```
   NewSet := OldSet;
   ```

   In this case, NewSet is an exact copy of OldSet.

3. You may declare an array of sets, as in:

```
DaysArray = ARRAY[1..10] OF DaySet;
```

   where DaySet is a previously declared set type.

4. You cannot read or write a set using the input/output statements, but there are some programming techniques using set operations (explained in the next section) that may be used.
5. Sets can be passed as parameters of subprograms, in which case they must be declared as types.

8-3 SET OPERATORS AND OPERATIONS

In addition to the membership operator "IN," you can use the following operations with sets of compatible types:

1. Union (+)
2. Intersection (*)
3. Difference (–)

UNION The union of two sets "S1" and "S2" is a set whose elements are in either S1 or S2, or in both. The operator "+" is used for this operation, for example (using the previous declarations):

```
Alphabet := Small + Capital;
```

The set "Alphabet" will thus contain both the lowercase and the uppercase letters.

INTERSECTION The intersection of two sets results in a set whose members are the elements common to both sets. For example, the statement:

```
MLL := [C,CPP,Cobol] * [Basic,Fortran,C,CPP]
```

results in the set MLL, which contains C and CPP.

DIFFERENCE The difference of two sets "S1" and "S2" is the set whose members are in S1 but not in S2. For example, the statement:

```
HLL := ProgCodes - LLL;
```

results in the set HLL, which contains all the elements of the set "ProgCodes" except "Assembly."

DRILL 8-1

Evaluate the following expressions:

(See the answers in the file DRL8-1.TXT on the distribution disk.)

```
1. ['A','B','C','D'] + ['E','F']
2. ['A','B','C','D'] + ['B','C','E','F']
3. [1,3,7] + []
4. ['A','D','F'] * ['O','F']
5. [1,2,3,4] * [5,6,7]
6. [1,2,3,4] - [5,6,7]
7. [5,6,7] - []
8. [Able, Baker, Charlie] - [Able, Charlie]
```

You can make use of the union operation to construct a new set (read a set) by reading one element at a time and adding it to the set, for example:

```
Read(NewElement);
Set1 := Set1 + [NewElement];
```

You can also make use of the difference operation to display the elements of a set. This is done by testing the membership of a variable of the same base type as that of the set. If the element is a set member, it is displayed, subtracted from the set, and replaced by its successor. This continues until the set is empty.

RELATIONAL OPERATORS The relational operators =, >=, <=, and <> can be used with sets of compatible types.

The meanings of the set relational operators are indicated in table 8-1 by comparing two sets "S1" and "S2." The table contains TRUE expressions as examples of each operation.

The operators > and < are not mentioned in the table as they may not be used with sets.

Table 8-1 Sets Relational Operators

Expression	*Meaning*	*Example*
S1 = S2	Both S1 and S2 contain the same elements.	[1,0] = [1,0]
S1 <> S2	S1 and S2 do not contain the same elements.	[1,0] <> [1,4]
S1 >= S2	All elements of S2 are in S1.	[1,2,3,4] >= [1,2] [1,2,3] >= [1,2,3]
S1 <= S2	All elements of S1 are in S2.	[] <= [1,2,3] [1,2,3] <= [1,2,3]

The relative precedence of Pascal operators (including the new operator "IN") is shown in table 8-2. Notice that the set operators (+, -, *) use the same symbols as the arithmetic operators. Also, the relational operators are used with either simple data types or sets.

Table 8-2 Precedence of Pascal Operators

Operator	*Precedence*
NOT	Priority 1 (highest)
* / DIV MOD AND	Priority 2
+ - OR (XOR in Turbo Pascal)	Priority 3
= > < >= <= <> IN	Priority 4 (lowest)

You may combine relational expressions using the boolean operators AND, OR, and NOT, but you must watch the precedence of operators, for example:

```
IF (Ch IN Small) AND (Ch IN Capital) THEN ...
```

The parentheses are necessary in this expression because the IN operator has a lower precedence than the AND operator.

DRILL 8-2

Write a program to test the expressions in table 8-1.

EXAMPLE: TEXT ANALYZER In the following program, the printable characters are divided into the following sets:

1. Lowercase letters
2. Uppercase letters
3. Alphabetic characters (which is the union of 1 and 2)
4. Digits
5. Punctuation characters
6. Other characters

The program reads a text file from the keyboard character by character and tests each character to see if it is a member in any one of these sets. The program is straightforward and contains four parts: declarations of sets, initialization of counters, testing memberships of characters, and displaying results.

```
{ ------------------------------ figure 8-1 ------------------------------ }
PROGRAM TextAnalyzer(INPUT,OUTPUT);
TYPE
   LowerCase  = SET OF 'a'..'z';
   UpperCase  = SET OF 'A'..'Z';
   Digits     = SET OF '0'..'9';
   Characters = SET OF CHAR;
VAR
   Capital                          :UpperCase;
   Small                            :LowerCase;
   Numerals                         :Digits;
   Alphabet, Punctuation, Others    :Characters;
   A, C, S, N, P, O, Counter        :INTEGER;
   Ch                               :CHAR;
BEGIN
   Counter := 0;  { counter of all characters }
   A := 0;        { counter of alphabetic characters }
   C := 0;        { counter of capital letters }
   S := 0;        { counter of small letters }
   N := 0;        { counter of numeric characters }
   P := 0;        { counter of punctuation characters }
   O := 0;        { counter of other characters }
   Small := ['a'..'z'];
   Capital := ['A'..'Z'];
   Alphabet := Small + Capital;
   Numerals := ['0'..'9'];
   Punctuation := [',',';','-','''','.','!','?',')','(','"',':','_'];
   WRITELN('Start typing your text file. To terminate press Ctrl-Z:');
   WHILE NOT EOF DO
      BEGIN
         WHILE NOT EOLN DO
            BEGIN
               READ(Ch);
               Counter := Counter + 1;
               IF Ch IN Alphabet THEN
                  BEGIN
                     A := A + 1;
                     IF Ch IN Small THEN
                        S := S + 1
                     ELSE IF Ch IN Capital THEN
                        C := C + 1
                  END
               ELSE IF Ch IN Numerals THEN
                  N := N + 1
```

```
            ELSE IF Ch IN Punctuation THEN
              P := P + 1
            ELSE
              O := O + 1
          END;
        READLN
      END;
  WRITELN('Total number of characters        = ', Counter);
  WRITELN('Number of alphabetic characters   = ', A);
  WRITELN(' .Number of lowercase letters: ', S);
  WRITELN(' .Number of uppercase letters: ', C);
  WRITELN('Number of numeric characters      = ', N);
  WRITELN('Number of punctuation characters  = ', P);
  WRITELN('Number of other characters        = ', O)
END.
```

Sample run:

```
Start typing your text file. To terminate press Ctrl-Z:
The standard set operators are:
        1. Union (+).
        2. Intersection (*).
        3. Difference (-).
^Z                        ----> Press Ctrl-Z to end the text
Total number of characters        = 85
Number of alphabetic characters   = 53
 .Number of lowercase letters: 49
 .Number of uppercase letters: 4
Number of numeric characters      = 3
Number of punctuation characters  = 14
Number of other characters        = 15
```

Sets are useful for testing conditions. One common use of sets is to precede a CASE statement in order to filter out the unwanted data which do not belong to any case.

8-3 RECORDS

A record, another structured type in Pascal, is a collection of related data items which may be of different types. Each item in the record is called a *field*. Take a look at this record, which is used to store information about each employee in a company:

Field #	Employee Record Information	Possible Data Type
1.	Name	STRING
2.	Address	STRING
3.	Phone number	STRING/INTEGER
4.	Hourly rate	REAL
5.	Marital status	CHAR/Enumeration

Unlike arrays (which contain elements of the same type) records may contain fields of any data type, including the type RECORD itself.

RECORD DECLARATION The declaration of a record takes the form:

```
type-identifier = RECORD
                    field-list
                  END;
```

The field list contains the name and type of each field as in this declaration of the record "EmployeeRecord."

```
TYPE
   EmployeeRecord = RECORD
                      Name    :STRING[25];
                      Address :STRING[40];
                      Phone   :STRING[12];
                      Rate    :REAL;
                      MaritalStatus:CHAR;
                    END;
```

(Note: If your Pascal implementation does not support the STRING type, you may replace the STRING variables by INTEGER or CHAR variables, in which case you need to replace the variable "Name" with another variable like "ID," etc.)

A record declaration must be terminated by the keyword END.

In the VAR section, the record is then declared as a variable of the type "EmployeeRecord":

```
VAR
   EmployeeRec :EmployeeRecord;
```

As with other structured and user-defined data types, you can declare a record in the VAR section directly, but you now know the advantages of declaring data structures as types.

ACCESSING FIELDS Each field in a record can be accessed using both the record identifier and the field identifier separated by a period. For example, you can assign values to the fields with statements like:

```
EmployeeRec.Name := 'Charles A. Dixon';
EmployeeRec.Rate := 22.5;
```

You can do the same thing with input and output operations:

```
WRITELN('Employee Name: ', EmployeeRec.Name);
```

This type of compound variable is called a *fielded variable*. Actually, the scope of the field identifier (such as "Name") is the record in which it was declared, and it may be used elsewhere in the program as the name of another variable if desired.

In the following example, the record "EmployeeRec" is filled and then displayed.

```
{-------------------------------- figure 8-2 ------------------------------}
PROGRAM RecordExample1(OUTPUT);
TYPE
  EmployeeRecord = RECORD
                     Name:STRING[25];
                     Address:STRING[40];
                     Phone:STRING[12];
                     Rate:REAL;
                     MaritalStatus:CHAR;
               END;
VAR
  EmployeeRec :EmployeeRecord;
BEGIN
{ Assign values to the fielded variables }
  EmployeeRec.Name := 'Diane J. Bedford';
  EmployeeRec.Address := '20 Carmen Avenue, New Orleans, LA 70112';
  EmployeeRec.Phone := '504-666-5043';
  EmployeeRec.Rate := 28.5;
  EmployeeRec.MaritalStatus := 'S';
{ Display record information }
  WRITELN('Employee Name:    ', EmployeeRec.Name);
  WRITELN('Address:          ', EmployeeRec.Address);
  WRITELN('Telephone #:      ', EmployeeRec.Phone);
  WRITELN('Hourly Rate:     $', EmployeeRec.Rate:0:2);
  WRITELN('Marital Status:   ', EmployeeRec.MaritalStatus)
END.
```

The output is:

```
Employee Name:  Diane J. Bedford
Address:        20 Carmen Avenue, New Orleans, LA 70112
Telephone #:    504-666-5043
Hourly Rate:    $28.50
Marital Status: S
```

THE WITH STATEMENT There is, however, a shorter way to do this by using the WITH statement. When you use the WITH statement, you do not have to use fielded variables. Look at this block of assignments using the WITH statement:

```
WITH EmployeeRec DO
  BEGIN
    Name := 'Charles A. Dixon';
    Address := '202 Greenwood, Gretna, LA 70088';
    Phone := '504-666-7574';
    Rate := 22.5;
    MaritalStatus := 'M'
  END;
```

The effect of using the WITH statement is to attach each field name to the record name. If one of the variables inside the block is not a field identifier, it will not be modified by the WITH statement. If WITH is followed by only one statement, there is of course no need for the BEGIN-END block.

You can use the WITH statement to call a procedure to process the fields of a record, for example:

```
WITH EmployeeRec DO
  DisplayResults(Name, Rate);
```

This statement is equivalent to:

```
DisplayResults(EmployeeRec.Name, EmployeeRec.Rate);
```

The WITH statement takes the general form:

> **WITH** record-identifier **DO**
> statement;

The following example demonstrates the same logic as that used in program 8-2, but the program is divided into three subprograms: "GetData," "DisplayInfo," and "DrawLine" (which you wrote before). The output of this program is displayed in the proper format, using a header for the record.

```
{------------------------------- figure 8-3 -------------------------------}
PROGRAM RecordExample2(OUTPUT);
TYPE
   EmployeeRecord = RECORD
                         Name:STRING[25];
                         Address:STRING[40];
                         Phone:STRING[12];
                         Rate:REAL;
                         MaritalStatus:CHAR;
                    END;
VAR
   EmployeeRec    :EmployeeRecord;
{ ------------ Procedure Drawline -------------- }
PROCEDURE DrawLine(LineLength, TabLength :INTEGER);
CONST
   Dash = '-';
VAR
   Counter :INTEGER;
BEGIN
   FOR Counter := 1 TO TabLength DO
      WRITE(' ');
   FOR Counter := 1 TO LineLength DO
      WRITE(Dash);
   WRITELN
END;
{ ------------- Procedure GetData -------------- }
PROCEDURE GetData(VAR Employee :EmployeeRecord);
{Name,Address,Phone,Rate,MaritalStatus);}
{ Assign values to fields }
BEGIN
   WITH Employee DO
      BEGIN
     Name := 'Diane J. Bedford';
     Address := '20 Carmen Avenue, New Orleans, LA 70112';
     Phone := '504-666-5043';
     Rate := 28.5;
     MaritalStatus := 'S'
      END
END;
{ ---------- Procedure DisplayInfo ------------ }
PROCEDURE DisplayInfo(Employee :EmployeeRecord);
{ Display record information }
CONST
   Header ='Record of ';
```

```
VAR
   Len, Tab, Counter  :INTEGER;
   HeaderText, Status :STRING;
BEGIN
   WITH Employee DO
      BEGIN
         HeaderText := CONCAT(Header,Name);
         Len := LENGTH(HeaderText);
         Tab := (80- Len) DIV 2;
         DrawLine(Len, Tab);
         FOR Counter := 1 TO Tab DO
            WRITE(' ');
         WRITELN(HeaderText);
         DrawLine(Len, Tab);
         WRITELN('Address:           ', Address);
         WRITELN('Telephone #:       ', Phone);
         WRITELN('Hourly Rate:      $', Rate:0:2);
         IF MaritalStatus = 'M' THEN
            Status := 'Married'
         ELSE
            Status := 'Single';
         WRITELN('Marital Status:  ', Status)
      END
END;
{ --------------- Main Program ----------------- }
BEGIN
   GetData(EmployeeRec);
   DisplayInfo(EmployeeRec)
END.
```

The output is:

```
                           --------------------------
                           Record of Diane J. Bedford
                           --------------------------
Address:          20 Carmen Avenue, New Orleans, LA 70112
Telephone #:      504-666-5043
Hourly Rate:      $28.50
Marital Status:   Single
```

The points which are worthy of your attention in the program are the use of the WITH statement and the passing of the record as a parameter to the subprograms. Notice also that the record is passed once as a variable parameter (using VAR), when it was to return values of the fields, and once as a value parameter, when it was only a receiver.

The actual value of such a program comes when it reads the employee information from a data file, which will be discussed shortly.

8-4 NESTING RECORDS

In the EmployeeRecord example you may split the field address information to street address, city, state, and zip code. This means that the address field becomes a record nested in the EmployeeRecord. The new record will look as follows:

```
TYPE
   AddressRecord = RECORD
                  Street:STRING[18];
                  City:STRING[15];
                  State:STRING[2];
                  Zip :String[5];
                    END;
   EmployeeRecord = RECORD
                  Name     :STRING[25];
                  AddressRec:AddressRecord;
                  Phone    :STRING[12];
                  Rate     :REAL;
                  MaritalStatus:CHAR;
                END;
VAR
   EmployeeRec     :EmployeeRecord;
```

In this declaration, you have two record types: "AddressRecord" and "EmployeeRecord." The field "AddressRec" in the employee record is of the type "AddressRecord" which was defined before. To deal with any fielded variables in the "AddressRec" you have to attach both names of the two records "EmployeeRec" (which is the grandparent) and "AddressRec" (which is the parent). Here are some sample assignments:

```
EmployeeRec.AddressRec.Street := '15 Darell Street';
EmployeeRec.AddressRec.Zip := '60108';
```

When you display any of these fields you use the same method:

```
WRITELN(EmployeeRec.AddressRec.Street);
WRITELN(EmployeeRec.AddressRec.City);
```

Here is the complete program:

```
{ ----------------------------- figure 8-4 ----------------------------- }
PROGRAM NestedRecord(OUTPUT);
TYPE
   AddressRecord   = RECORD
                        Street:STRING[18];
                        City:STRING[15];
                        State:STRING[2];
                        Zip:String[5];
                     END;
   EmployeeRecord = RECORD
                       Name:STRING[25];
                       AddressRec:AddressRecord;
                       Phone:STRING[12];
                       Rate:REAL;
                       MaritalStatus:CHAR;
                  END;
VAR
   EmployeeRec     :EmployeeRecord;
BEGIN
   EmployeeRec.Name := 'Jean L. Krauss';
   EmployeeRec.AddressRec.Street := '15 Darell Street';
   EmployeeRec.AddressRec.City := 'Bloomingdale';
   EmployeeRec.AddressRec.State := 'IL';
   EmployeeRec.AddressRec.Zip := '60108';
   EmployeeRec.Phone := '312-987-5432';
   EmployeeRec.Rate := 27.5;
   EmployeeRec.MaritalStatus := 'M';
   WRITELN('Employee Name:   ', EmployeeRec.Name);
   WRITELN('Address:         ', EmployeeRec.AddressRec.Street);
   WRITELN('                 ', EmployeeRec.AddressRec.City);
   WRITE('                 ', EmployeeRec.AddressRec.State);
   WRITELN(' ', EmployeeRec.AddressRec.Zip);
   WRITELN('Telephone #:     ', EmployeeRec.Phone);
   WRITELN('Hourly Rate:     $', EmployeeRec.Rate:0:2);
   WRITELN('Marital Status:  ', EmployeeRec.MaritalStatus)
END.
```

The output is:

```
Employee Name:   Jean L. Krauss
Address:         15 Darell Street
                 Bloomingdale
                 IL 60108
```

```
Telephone #:      312-987-5432
Hourly Rate:      $27.50
Marital Status:   M
```

If you would like to use the WITH statement with such a nested record you need two nested WITH blocks, thus:

```
WITH EmployeeRec DO
   WITH AddressRec DO
      BEGIN
      Name := 'Tammy M. Ockman';
      Street := '344 Temple Dr.';
      ...
      END;
```

If any field identifier belongs to the AddressRec, it will be modified by both AddressRec and EmployeeRec, but if it belongs to the EmployeeRec directly, it will be modified by EmployeeRec only. If it is a regular variable, it will not be modified at all.

DRILL 8-3

Write the complete program that initializes and displays the employee record using the WITH statement with the nested address record shown above.

SUMMARY

In this chapter you have met two structured data types, the set and the record, and are now familiar with their features.

1. You now know how to declare a set of a specific base type using the form:

   type-identifier = **SET OF** base-type;

2. You also know the standard set operators (union (+), intersection (*), and difference (–)) and the set relational operators (= >= <= <>), and learned how to use these operators to process sets.
3. You are familiar with restrictions on sets, as well as their main uses in programming.
4. You can declare record types using the form

   RECORD
   filed-list
   END;

5. You can access fields using either fielded variables or the **WITH** statement, which takes the form:

   WITH record-identifier **DO**
   statement;

6. You also know how to declare and use nested records and how to process them as well.

CHAPTER NINE

FILES AND APPLICATIONS

9-1 DATA FILES

In the previous chapters you have used different data structures in which to store data items, and you know how to organize your data for optimum processing efficiency. If you do not store data to the disk, however, every data item you entered into a program will evaporate when the program exits. Using disks to store your data in files will enable you to save your data permanently and retrieve them later for either reviewing or further processing.

A FILE (which is a structured type) is in general defined as a collection of related items stored on disk or any other external storage medium and arranged in sequence as shown in the following figure.

item-1	item-2	item-3	item-4	...	EOF

A file item (also called a file *component* or file element) may be of any simple or structured type except the type FILE.

Files may be accessed to perform either one of the following operations:

- Reading from a file (input)
- Writing to a file (output)

Files can be organized as either *sequential* access files or *random (direct)* access files. In the first method any item in the file cannot be reached unless all the preceding items are read. One example of a sequential access file is a purchase list which has to be read from the top down to access a specific item. The random access file is organized like a set of post office boxes, which are

identified by numbers and accessed directly without the need to go through them all.

While standard Pascal allows only sequential access files, many implementations of Pascal (including Turbo and UCSD) provide random access files as well. The files discussed in this chapter are all sequential access files.

9-2 TEXT FILES

Standard Pascal provides two types of files, TEXT files and non-TEXT files (also called *binary* or *typed files*).

A TEXT file is the simpler file structure as its elements are all characters (of the type CHAR). You have already used the standard INPUT file (the keyboard) and the standard OUTPUT file (the screen), which are classified as TEXT files. A TEXT file consists of successive lines of characters separated by End-Of-Line marks and ends with the End-Of-File mark, as in this example:

```
This is a text file. (EOLN)
Each line is composed of successive characters. (EOLN)
Lines are separated by End-Of-Line marks. (EOLN)
The file is terminated by an End-Of-File mark. (EOLN)
(EOF)
```

The file on the disk looks exactly the same as the file you type onto the screen from the keyboard. The characters in a TEXT file are stored in the ASCII format (or EBCDIC in some systems), which means that if the file contains a number like "1234," it will be stored in four bytes, each byte representing the ASCII code of a digit. This is not the case if the number is treated as an INTEGER, in which case it is stored in the internal binary format (0000010011010010) in two bytes.

9-3 READING A TEXT FILE

In chapter eight you used program 8-1 to read a TEXT file from the keyboard character by character and categorize each character in the file. In this section, the same logic will be used to read and analyze a previously stored text file on the disk. You need to make a few changes in program 8-1 to make it read a disk file. As we discuss these changes, you will learn the protocols necessary to retrieve information from a TEXT file.

THE FILE VARIABLE In order to use a disk TEXT file you have to declare a *file variable* of the type TEXT. If you choose a name like "DiskFile" for this purpose, the declaration will be:

```
VAR
   DiskFile :TEXT;
```

FILE PARAMETERS To use a file in standard Pascal, you must include the file variable ("DiskFile" in our example) among the other file parameters in the program header (this is not necessary in other implementations).

```
PROGRAM TextAnalyzer2(OUTPUT, DiskFile);
```

Here, the parameter INPUT is not needed as long as no data are to be read from the keyboard. The parameter OUTPUT, however, is necessary for displaying the output.

OPENING A FILE FOR INPUT: RESET To read a text file, it must be *opened* using the standard procedure RESET as follows:

```
RESET(DiskFile);
```

The parameter of the procedure is the file variable.

In Turbo Pascal another procedure ASSIGN must also be used to link the actual data file on the disk to the file variable. If the text file to be read is the file C:\CONFIG.SYS (which already exists in the root directory of the hard disk C:), then the following statement must be used before opening the file:

```
ASSIGN(DiskFile, 'C:\CONFIG.SYS');
```

You may replace the file CONFIG.SYS with any other existing file, or you can write a new text file with any text editor (such as EDIT or EDLIN). In any case, if the file is not in the current directory or on the current disk, you have to include the complete PATHNAME of the file as shown in the statement above (for more details on PATHNAME refer to your DOS manual).

Because Turbo Pascal was used to compile the programs in this book, the following two statements were used to open the file:

```
ASSIGN(DiskFile, 'C:\CONFIG.SYS');
RESET(DiskFile);
```

Now the file "config.sys" is ready for input, and the file pointer is pointing to the first item (character) in the file.

Some implementations (such as UCSD) link the file variable and the file name with the RESET procedure, thus making the ASSIGN procedure unnecessary:

```
RESET(file-variable, file-name);
```

CLOSING THE FILE The last step after you are finished with reading or writing to a disk file is to close it using the CLOSE procedure, or else data will be lost. This procedure is neither available nor necessary in standard Pascal, where punched cards and magnetic tape files were used.

To close a file, the procedure CLOSE is used as in:

```
CLOSE(DiskFile);
```

Some versions use more parameters for the CLOSE procedure. For example, in UCSD the file must be closed after writing to it using the form:

```
CLOSE(file-variable, action);
```

where "action" is replaced by either the keyword LOCK if the file will be retained, or PURGE if the file will be deleted.

To summarize, these are the general formulas for preparing a text file for input:

- Program header:
  PROGRAM Program-name(file-list);

(The file-list is optional in most versions.)

- File variable declaration:
  file-variable :TEXT;
- Linking file variable to file name (Turbo only):
  ASSIGN(file-variable, file-name);
- Opening a file for input:
  RESET(file-variable);
- Closing a file (for versions other than standard Pascal):
  CLOSE(file-variable);
  CLOSE(file-variable, action); (UCSD)

FILE INPUT PROCEDURES: READ, READLN The input/output statements you have used before are actually special cases of the general form. The complete form of the READLN (or READ) procedure is:

READLN(file-variable, input-list);
READ(file-variable, input-list);

If no file-variable is used, the form is reduced to the one you have been using:

READLN(input-list);

This is the same as using the name of the standard **INPUT** file:

READLN(INPUT, input-list);

Now, in our example we are going to use the file-variable "Diskfile." The input statement will be:

```
READ(DiskFile,Ch);
```

where Ch is the character variable to be read.

THE EOF AND EOLN FUNCTIONS The general form of the EOLN function includes the file variable as follows:

EOLN(file-variable)

If the file-variable and parentheses are omitted, the standard INPUT file (the keyboard) is assumed.

The same thing goes for the EOF function:

EOF(file-variable)

APPLICATION 1: DISK-FILE TEXT ANALYZER Now that you have all the tools for reading a file you can examine the following program, which will give you a complete report on the text file CONFIG.SYS in your computer. Do not expect to get the same result as the one obtained from this sample run, because different computers may have different configuration files.

Following the program, the CONFIG.SYS file is listed to check the validity of the report.

```
{ ---------------------------- figure 9-1 ---------------------------- }
PROGRAM TextAnalyzer2(OUTPUT,DiskFile);
{ Reading from a disk text file one character at a time }
TYPE
```

```
   LowerCase  = SET OF 'a'..'z';
   UpperCase  = SET OF 'A'..'Z';
   Digits     = SET OF '0'..'9';
   Characters = SET OF CHAR;
VAR
   DiskFile                        :TEXT;      { declare a text file variable }
   Capital                         :UpperCase;
   Small                           :LowerCase;
   Numerals                        :Digits;
   Alphabet, Punctuation, Others   :Characters;
   A, C, S, N, P, O, Counter       :INTEGER;
   Ch                              :CHAR;
BEGIN
{ Link the file variable to the file name 'C:\CONFIG.SYS' }
   ASSIGN(DiskFile, 'C:\CONFIG.SYS');
{ Open the file for input }
   RESET(DiskFile);
{ The program logic }
   Counter := 0;  { counter of all characters }
   A := 0;        { counter of alphabetic characters }
   C := 0;        { counter of capital letters }
   S := 0;        { counter of small letters }
   N := 0;        { counter of numeric characters }
   P := 0;        { counter of punctuation characters }
   O := 0;        { counter of other characters }
   Small := ['a'..'z'];
   Capital := ['A'..'Z'];
   Alphabet := Small + Capital;
   Numerals := ['0'..'9'];
   Punctuation := [',',';','-','''','.','!','?',')','(','"',':','_'];
{ Check for the end of the disk file }
   WHILE NOT EOF(DiskFile) DO
      BEGIN
{ Check for the end of line in the disk file }
         WHILE NOT EOLN(DiskFile) DO
            BEGIN
{ Read one character from the disk file }
               READ(DiskFile,Ch);
               Counter := Counter + 1;
               IF Ch IN Alphabet THEN
                  BEGIN
                     A := A + 1;
                     IF Ch IN Small THEN
                        S := S + 1
```

```
                    ELSE IF Ch IN Capital THEN
                         C := C + 1
                    END
                  ELSE IF Ch IN Numerals THEN
                    N := N + 1
                  ELSE IF Ch IN Punctuation THEN
                    P := P + 1
                  ELSE
                    O := O + 1
                END;
{ Advance the pointer to the next line }
        READLN(DiskFile)
      END;
{ End of the file is reached }
{ Close the file }
   CLOSE(DiskFile);
{ Display the report }
   WRITELN;
   WRITELN('Total number of characters          = ', Counter);
   WRITELN('Number of alphabetic characters     = ', A);
   WRITELN(' .Number of lowercase letters: ', S);
   WRITELN(' .Number of uppercase letters: ', C);
   WRITELN('Number of numeric characters        = ', N);
   WRITELN('Number of punctuation characters    = ', P);
   WRITELN('Number of other characters          = ', O);
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

```
Listing of the file "CONFIG.SYS":
DEVICE=C:\WINDOWS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE 4096 RAM
lastdrive=G
DEVICEHIGH=C:\DOS\SETVER.EXE
DOS=HIGH,UMB
FILES=25
BUFFERS=25
```

The program output is:

```
Total number of characters       = 129
Number of alphabetic characters  = 96
 .Number of lowercase letters: 9
 .Number of uppercase letters: 87
```

```
Number of numeric characters     = 11
Number of punctuation characters = 7
Number of other characters       = 15
```

9-4 DISPLAYING A TEXT FILE

You can display the contents of any text file by using the same logic as in the previous program, but adding a WRITE statement after each read:

```
READLN(DiskFile,Ch);
WRITE(Ch);
```

You also need to advance one line on the screen using a WRITELN statement whenever the EOLN is detected, or else the separate lines will be joined together.

Here is the program, which reads the same file (CONFIG.SYS). The name of the file is declared as a constant, and you may replace it with any file name. It is also possible to use the program source file itself (its name is 9-2.PAS on the distribution disk), in which case the program reads itself.

```
{ ------------------------------ figure 9-2 ------------------------------ }
PROGRAM ReadTextFile(INPUT,OUTPUT,DiskFile);
{ Reading a text file stored on the disk }
CONST
{ You may replace the following constant by any existing file name }
  FileName = 'C:\CONFIG.SYS';
VAR
  DiskFile :TEXT;
  Ch       :CHAR;
BEGIN
  ASSIGN(DiskFile, FileName);
  RESET(DiskFile);
  WHILE NOT EOF(DiskFile) DO
    BEGIN
      WHILE NOT EOLN(DiskFile) DO
        BEGIN
{ Read and display one character from the text file }
          READ(DiskFile,Ch);
          WRITE(Ch)
        END;
{ Advance the pointer to the next line }
      READLN(DiskFile);
{ Advance one line on the screen }
```

```
      WRITELN
    END;
  CLOSE(DiskFile);
  WRITELN('Press ENTER to continue..');
  READLN
END.
```

The output may look something like this:

```
DEVICE=C:\WINDOWS\HIMEM.SYS
DEVICE=C:\DOS\EMM386.EXE 4096 RAM
lastdrive=G
DEVICEHIGH=C:\DOS\SETVER.EXE
DOS=HIGH,UMB
FILES=25
BUFFERS=25
Press ENTER to continue..
```

READING A TEXT FILE AS A SET OF STRINGS If your version of Pascal supports the STRING type, you may read a TEXT file one line at a time.

The following program deals with the file as made of strings rather than characters. Each string has a maximum length of 80 characters, which is the expected line length. After each line is read the file pointer moves to the next line. If any line contains less than 80 characters, the dynamic length of the string will be set to the actual number of characters in the line. If on the other hand a line contains more than 80 characters, the rest are ignored. When you run the program it asks you to enter the name of the file to be displayed, so this program acts like the DOS command TYPE.

```
{ ------------------------------ figure 9-3 ------------------------------ }
PROGRAM DisplayTextFile(OUTPUT,MyFile);
{ Reading a text file stored on the disk one line at a time }
VAR
  MyFile             :TEXT;
  OneLine, FileName  :STRING[80];
BEGIN
  WRITE('Please enter the file name to be displayed: ');
  READLN(FileName);
  WRITELN;
  WRITELN('The contents of the file ',FileName,' are: ');
  ASSIGN(MyFile, FileName);
  RESET(MyFile);
{ Check for the end of the text file }
  WHILE NOT EOF(MyFile) DO
```

```
      BEGIN
{ Read and display the text file one line at a time }
         READLN(MyFile,OneLine);
         WRITELN(OneLine);
      END;
   CLOSE(MyFile);
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

If the file does not exist or its name is written incorrectly, the program gives an error message like this:

```
Please enter the file name to be displayed: C:\CNFIG.SYS
The contents of the file C:\CNFIG.SYS are:
Runtime error 002 at 0000:00F2.
```

Notice that a READLN statement was used to read each string. If you used a READ statement you would still have to use another READLN to skip over the End-Of-Line mark at the end of each line and move the file pointer to the beginning of the next line. This is because when you use the READ statement, it will read the string characters until the End-Of-Line mark (or a CR) is detected, then stop. It also does not move the pointer.

In this program you may check the EOLN after each read as you did when you read characters, but you do not need to.

READING MULTIPLE STRINGS It is possible to read more than one string with only one READLN (or READ), but this is sometimes iffy. To understand the possible pitfalls, take a look at this example which reads three strings, each of them declared as STRING[5], from a text file named "test.txt." This file contains the following line:

"This is a test text file."

```
{ ------------------------------ figure 9-4 ------------------------------ }
PROGRAM ReadMultipleStrings1(OUTPUT,F);
VAR
   F              :TEXT;
   Str1,Str2,Str3 :STRING[5];
BEGIN
   ASSIGN(F,'test.txt');
   RESET(F);
   READLN(F,Str1,Str2,Str3);
   WRITELN('Str1= ', Str1);
```

```
   WRITELN('Str2= ', Str2);
   WRITELN('Str3= ', Str3);
   CLOSE(F);
   WRITELN('Press ENTER to continue..');
   READLN
END.
```

The output is:

```
Str1= This
Str2= is a
Str3= test
```

As you can see in the output, each string variable is assigned five characters (including the blank spaces). Now replace the declaration of the string variables with the following:

```
Str1,Str2,Str3  :STRING;
```

If you run the program using this declaration, the length of each string will default to the maximum length supported by the language, and you will get the result:

```
Str1= This is a test text file.
Str2=
Str3=
```

What happened here was, the first variable was assigned the whole line (up to the End-Of-Line mark) and nothing was left for the other two. In short, you can only read multiple strings safely if you know the length of each one.

9-5 CREATING A TEXT FILE: REWRITE

To create a file you have to open the file to receive output. The procedure REWRITE (which is the counterpart of RESET) is used for this purpose. It takes the form:

REWRITE(file-variable);

In Turbo Pascal you have to link the file variable to the actual file name on the disk using ASSIGN as you did with input.

Some implementations (such as UCSD) instead use a modified formula of the procedure REWRITE, where both the file variable and the file name are used:

```
REWRITE(file-variable, file-name);
```

The rules of inventing a file name (which is the actual name of the disk file) depend on the operating system. In DOS the name can be made of up to eight characters and an optional extension of up to three characters (such as EMPLOYEE.DAT). After this statement an empty file is open and ready for writing.

NOTE If you open an existing file for output, the data in this file will be lost and overwritten by the new data.

THE OUTPUT PROCEDURES: WRITE, WRITELN To write one or more items to a file use the general form of the WRITELN (or WRITE) procedure:

WRITELN(file-variable, output-list);
or **WRITE**(file-variable, output-list);

If the file variable is omitted from these formulas, the standard OUTPUT file (the screen) is assumed and the form is reduced to the one you have been using:

WRITELN(output-list);

which is equivalent to:

WRITELN(OUTPUT, output-list);

After you are finished writing to a disk file you must close it with the CLOSE procedure as mentioned before.

In the following example a file HELLO.TXT is created, then the constant Hello Pascal is written to this file.

```
{ ------------------------------ figure 9-5 ---------------------------- }
PROGRAM CreateFile(F);
CONST
   TestSentence = 'Hello Pascal';
VAR
   F :TEXT;
BEGIN
   ASSIGN(F, 'HELLO.TXT');  { Turbo only }
   REWRITE(F);              { open the file for output }
   WRITELN(F, TestSentence);
   CLOSE(F)
END.
```

When this program is executed a new file HELLO.TXT is added to your current directory. In order to be sure that the file was written properly, you can

display it using either the DOS command TYPE or program 9-3 (which replaces it). In either case you will see the two words Hello Pascal on the screen.

As mentioned earlier, a text file can be created and written to with any text editor, but the importance of creating a file with a Pascal program comes when the information in the new file is based on data processed from other files.

DRILL 9-1

Write a program to accept from the keyboard the name and/or ID number and the hours worked per month for each employee, and write the data to a file called TIMSHEET.EMP. The program should process the data for any number of employees.

APPLICATION 2: EMPLOYEE FILE In chapter eight you created an employee record to contain information about the name, address, wages, etc. of each employee. In the following program, you are going to write the employee record information to a disk file EMPFILE.TXT using the nested record structure. Take a look at the program first:

```
{ ------------------------------ figure 9-6 ---------------------------- }
PROGRAM CreateEmpFile(INPUT,OUTPUT,F);
TYPE
  AddressRecord   = RECORD
                        Street:STRING[18];
                        City:STRING[15];
                        State:STRING[2];
                        Zip:String[5];
                    END;
  EmployeeRecord = RECORD
                       ID:INTEGER;
                       Name:STRING[20];
                       AddressRec:AddressRecord;
                       Phone:STRING[12];
                       Rate:REAL;
                       MaritalStatus:CHAR;
                    END;
VAR
  F           :TEXT;{ The file variable }
  EmployeeRec :EmployeeRecord;
BEGIN
  ASSIGN(F, 'EMPFILE.TXT');
    REWRITE(F);
```

```
      WITH EmployeeRec DO
        WITH AddressRec DO
          BEGIN
            WRITE('Please enter Employee ID: ');  READLN(ID);
            WRITE('Employee Name: ');              READLN(Name);
            WRITE('Address: Street: ');            READLN(Street);
            WRITE('           City: ');            READLN(City);
            WRITE('          State: ');            READLN(State);
            WRITE('       Zip code: ');            READLN(Zip);
            WRITE('Phone Number: ');               READLN(Phone);
            WRITE('Hourly Rate: ');                READLN(Rate);
            WRITE('Marital Status (S/M): ');       READLN(MaritalStatus);
{ Store the information to the file }
            WRITELN(F, ID);
            WRITELN(F, Name);
            WRITELN(F, Street);
            WRITELN(F, City);
            WRITELN(F, State);
            WRITELN(F, Zip);
            WRITELN(F, Phone);
            WRITELN(F, Rate:0:2);
            WRITELN(F, MaritalStatus)
          END;
      CLOSE(F)
END.
```

Sample run:

```
Please enter Employee ID: 122
Employee Name: Tammy M. Ockman
Address: Street: 322 Temple Dr.
           City: New Orleans
          State: LA
       Zip code: 70112
Phone Number: 504-285-3434
Hourly Rate: 22.45
Marital Status (S/M): S
```

The following is a display of the file contents:

```
122
Tammy M. Ockman
322 Temple Dr.
New Orleans
```

```
LA
70112
504-285-3434
22.45
S
```

Notice that a numeric field "ID" has been added to the record, which is otherwise as before (in Chapter 8). Again, if your compiler does not support the STRING type (which is not likely), you can use the numeric and character fields only.

The resulting file contains as many lines as the number of fields in the record. Actually, you can write all of the fields in one line if you so wish by replacing the WRITELNs by WRITEs.

DRILL 9-2

Modify the last program so that it can store more than one employee record. You may wish to rebuild it as a procedure which can be called for each data entry of one employee.

APPLICATION 3: PAYROLL The file you have just created contains a good deal of information about employees and can be used for more than one purpose. You can use some or all of the information in this file to create different reports or other data files. In the following application, the file EMPFILE.TXT is read but only three fields from each record are used: "ID," "Name," and "HourlyRate." The program first displays an employee's information on the screen, then the user is prompted to enter "HoursWorked" for this employee. The "Wages" are then calculated by multiplying "HourlyRate" and "HoursWorked." After processing each record the "ID," "Name," and "Wages" are stored in a new file PAYFILE.TXT. The new file is used to produce a payroll report for this pay period.

```
{ ------------------------------ figure 9-7 ---------------------------- }
PROGRAM PayRoll(INPUT,OUTPUT,MasterFile,PayFile);
TYPE
  AddressRecord  = RECORD
                     Street:STRING[18];
                     City:STRING[15];
                     State:STRING[2];
                     Zip:String[5];
                   END;
  EmployeeRecord = RECORD
                     ID:INTEGER;
```

```
                Name:STRING[20];
                AddressRec:AddressRecord;
                Phone:STRING[12];
                Rate:REAL;
                MaritalStatus:CHAR;
              END;
  PayRecord = RECORD
              ID:INTEGER;
              Name:STRING[20];
              Wages:REAL;
            END;
VAR
  MasterFile, PayFile:TEXT;
  EmployeeRec :EmployeeRecord;
  PayRec      :PayRecord;
  HoursWorked, Wages:REAL;

{ --------------- Procedure GetInfo ------------------ }
{ This procedure reads the employee file "EMPFILE.TXT"
  and displays the ID, Name, and Hourly Rate.        }
PROCEDURE GetInfo(VAR F:TEXT);
BEGIN
    WITH EmployeeRec DO
      WITH AddressRec DO
        BEGIN
          READLN(F,ID);       WRITELN('ID: ',ID);
          READLN(F,Name);     WRITELN('Name: ',Name);
          READLN(F,Street);
          READLN(F,City);
          READLN(F,State);
          READLN(F,Zip);
          READLN(F,Phone);
          READLN(F,Rate);     WRITELN('Hourly rate: $', Rate:0:2);
          READLN(F,MaritalStatus);
        END;
END;

{ --------------- Procedure CalcWages ---------------- }
{ This procedure is used to calculate wages.
  The result is returned to the main program }
PROCEDURE CalcWages(HoursWorked:REAL; VAR Wages:REAL);
BEGIN
    WITH EmployeeRec DO
      Wages := Hoursworked * Rate;
```

```
      Wages := ROUND(100 * Wages) / 100   { rounding cents }
END;

{ -------------- Procedure FilePayRoll --------------- }
{ This procedure is used to write one record to
  the output file "PAYFILE.TXT"                 }
PROCEDURE FilePayRoll(VAR F :TEXT; VAR P :TEXT; Wages :REAL);
BEGIN
      WITH EmployeeRec DO
         BEGIN
            PayRec.ID := ID;
            PayRec.Name := Name;
            Payrec.Wages := Wages
         END;
      WITH PayRec DO
         WRITELN(P, ID:3, Name:20, Wages:10:2)
END;

{ --------------- Main Program ------------------ }
BEGIN
   ASSIGN(MasterFile, 'EMPFILE.TXT');   RESET(MasterFile);
   ASSIGN(Payfile, 'PAYFILE.TXT');      REWRITE(PayFile);
   WHILE NOT EOF(MasterFile) DO
      BEGIN
         GetInfo(MasterFile);
         WRITE('Please enter hours worked for this pay period: ');
         READLN(HoursWorked);
         CalcWages(HoursWorked, Wages);
         FilePayRoll(MasterFile, PayFile, Wages)
      END;
   CLOSE(MasterFile);
   CLOSE(PayFile)
END.
```

Sample run:

Assume that the file EMPFILE.TXT contains three records. The program will use these records as follows:

```
ID: 122                              ----> Information from file
Name: Tammy M. Ockman                ----> Information from file
Hourly rate: $22.45                  ----> Information from file
Please enter hours worked for this pay period: 160   ----> Entered by user
ID: 123
Name: Tara S. Strahan
```

```
Hourly rate: $15.24
Please enter hours worked for this pay period: 160
ID: 125
Name: John G. Trainer
Hourly rate: $28.55
Please enter hours worked for this pay period: 140.5
```

The program creates the file PAYFILE.TXT containing the following records:

```
122     Tammy M. Ockman   3592.00
123     Tara S. Strahan   2438.40
125     John G. Trainer   4011.28
```

The program consists of three procedures:

1. "GetInfo" to read one record of the file EMPFILE.TXT and display only the selected fields. Notice that you have to read all of the record fields even if you do not need them all.
2. "CalcWages" to carry out the calculations.
3. "FilePayRoll" to write the record "PayRec" to the file PAYFILE.TXT.

These are some important points of the program:

A. When file variables (such as "MasterFile" and "PayFile") are passed to subprograms, they must be passed as variable parameters (using VAR). The type TEXT is used with such parameters:

```
PROCEDURE FilePayRoll(VAR F :TEXT; VAR P :TEXT; Wages :REAL);
```

B. Some identifiers (such as "Name" and "ID") are used in both "EmployeeRec" and "PayRec." This does not cause any problem because they are all fielded variables; remember that the scope of a fielded variable is limited to its own record. Also, the identifier "Wages" was declared both as a global variable and as a fielded variable (in the record "PayRec") and was also used as a local variable in the procedure "FilePayroll."

C. Take a look at these assignment statements in the procedure "FilePayRoll":

```
WITH EmployeeRec DO
  BEGIN
        PayRec.ID := ID;
        PayRec.Name := Name;
        Payrec.Wages := Wages
```

The first two statements copy the values of the fields "ID" and "Name" from "EmployeeRec" to the corresponding fields in "Payrec." The WITH statement

modifies only the variables which belong to the record "EmployeeRec" ("ID" and "Name"). A variable such as "PayRec.ID" is not affected by the WITH statement because it is explicitly modified by "Payrec." In the last statement, no variables at all are affected by the WITH statement.

DRILL 9-3

Add a procedure to the last program to display a Payroll Summary report as shown:

```
--------- PayRoll Summary ---------
ID --------- Name -------- Salary
122     Tammy M. Ockman   $3592.00
123     Tara S. Strahan   $2438.40
125     John G. Trainer   $4011.28
----------------------------------
```

The program may also be modified in such a way as to read the "HoursWorked" from the file TIMSHEET.EMP which you created in drill 9-1.

9-6 NON-TEXT FILES

The text file is a special predefined type of file, but as mentioned earlier the general definition of a file allows the file components to be of any type other than the type FILE. You can declare a file of any predefined or user-defined type using the form:

type-identifier = **FILE OF** component-type;

The component type can be a simple type (like INTEGER), a structured type (like an array), or a user-defined type (like a record).

The following is an example of a file declaration whose components are records (a simplified form of "EmployeeRecord" is used to make the program shorter):

```
TYPE
   EmployeeRecord = RECORD
                         ID:INTEGER;
                         Name:STRING[20];
                         Rate:REAL;
                    END;
   EmpFileRec = FILE OF EmployeeRecord;
VAR
```

```
    F          :EmpFileRec;          { The file variable }
    EmployeeRec :EmployeeRecord;     { The record variable }
```

The main properties of non-TEXT files are:

A. Data are represented in the internal binary format, which means that you cannot display the contents of a file using the DOS command TYPE. This also speeds up the transfer of data to and from the file.

B. The main advantage of non-TEXT files comes when using structured types such as arrays or records, because then you do not need to read or write the record field by field. For example, after the previous declarations you may read or write the whole record using these statements:

```
READ(F, EmployeeRec);
WRITE(F, EmployeeRec);
```

C. Because non-TEXT files are not made up of lines as TEXT files are, the procedures READLN and WRITELN may not be used with these files.

APPLICATION 4: PAYROLL SYSTEM This is the same payroll program but in a better shape. The program is divided into two separate modules (programs). The first module (figure 9-8) reads the employees' records from the keyboard and stores them in a non-TEXT file EMPFILE.BIN. In the second module (figure 9-9) the "HoursWorked" are entered from the keyboard and wages are calculated and written to the file PAYFILE.TXT, which is a TEXT file. The first program may be used only once to create the employee file, but the second program is used every pay period to create the "PayFile."

Here is the first program:

```
{ ------------------------------ figure 9-8 ------------------------------ }
PROGRAM EmpPayInfo(INPUT,OUTPUT,F);
{ This program is used to create a user-defined file 'EMPFILE.BIN'
  whose components are records. }
TYPE
  EmployeeRecord = RECORD
                     ID:INTEGER;
                     Name:STRING[20];
                     Rate:REAL;
                   END;
  EmpFileRec = FILE OF EmployeeRecord;
VAR
  F           :EmpFileRec;          { The file variable }
  EmployeeRec :EmployeeRecord;
{ ----------- Procedure WriteRecord ---------- }
```

```
PROCEDURE WriteRecord;
BEGIN
{ Store one record in the file }
  WRITE(F, EmployeeRec)
END;
{ ----------- Procedure GetData ---------- }
PROCEDURE getdata;
VAR
  Counter :INTEGER;
BEGIN
  Counter := 0;
  WITH EmployeeRec DO
    BEGIN
      WRITE('Please enter Employee ID (or 0 to end):'); READLN(ID);
      WHILE ID <> 0 DO
        BEGIN
          Counter := counter + 1;
          WRITE('Employee Name: ');  READLN(Name);
          WRITE('Hourly Rate: ');    READLN(Rate);
          WriteRecord;
          WRITE('Please enter Employee ID (or 0 to end):'); READLN(ID)
          END
      END;
  WRITELN(Counter, ' Employee records have been filed.')
END;
{ ---------------- Main Program -------------- }
{ Main Program }
BEGIN
  ASSIGN(F, 'EMPFILE.BIN');  REWRITE(F);
  GetData;
  CLOSE(F);
  WRITELN('Press ENTER to continue..');
  READLN
END.
```

The second module (PayRoll2) is made up of four procedures:

- "GetInfo" to read a record from the file EMPFILE.BIN.
- "CalcWages" to carry out the calculations.
- "FilePayroll" to write a record to the file PAYFILE.TXT.
- "ReadPayRoll" to read the file PAYFILE.TXT and display the payroll at the end of the process.

```
{ ----------------------------- figure 9-9 ------------------------------ }
PROGRAM PayRoll2(INPUT,OUTPUT,MasterFile,PayFile);
{ This program reads the file "EMPFILE.BIN" one record at a time,
  then calculates wages and stores the output in the text file
  "PAYFILE.TXT" }
TYPE
   EmployeeRecord = RECORD
                      ID   :INTEGER;
                      Name :STRING[20];
                      Rate :REAL;
                   END;
   PayRecord = RECORD
                 ID    :INTEGER;
                 Name  :STRING[20];
                 Wages :REAL;
              END;
   EmployeeFile = FILE OF EmployeeRecord;
VAR
   MasterFile          :EmployeeFile;
   PayFile             :TEXT;
   EmployeeRec         :EmployeeRecord;
   PayRec              :PayRecord;
   HoursWorked, Wages  :REAL;
{ -------------- Procedure GetInfo ----------------- }
{ This Procedure reads and displays a record from
the file "EMPFILE.BIN"                          }
PROCEDURE GetInfo(VAR F :EmployeeFile);
BEGIN
      READ(F,EmployeeRec);
      WITH EmployeeRec DO
         BEGIN
            WRITELN('ID: ',ID);
            WRITELN('Name: ',Name);
            WRITELN('Hourly rate: $', Rate:0:2);
         END;
END;
{ -------------- Procedure CalcWages --------------- }
PROCEDURE CalcWages(HoursWorked :REAL; VAR Wages :REAL);
BEGIN
      WITH EmployeeRec DO
         Wages := Hoursworked * Rate;
      Wages := ROUND(100 * Wages) / 100   { rounding cents }
END;
{ ------------- Procedure FilePayRoll -------------- }
```

```
{ This procedure writes a record to "PAYFILE.TXT"       }
PROCEDURE FilePayRoll(VAR F :EmployeeFile; VAR P :TEXT; Wages :REAL);
BEGIN
      WITH EmployeeRec DO
         BEGIN
            PayRec.ID := ID;
            PayRec.Name := Name;
            Payrec.Wages := Wages
         END;
      WITH PayRec DO
         BEGIN
            WRITELN(P, ID);
            WRITELN(P, Name);
            WRITELN(P, Wages);
         end;
END;
{ ------------- Procedure ReadPayRoll -------------- }
{ This procedure reads and displays "PAYFILE.TXT"        }
PROCEDURE ReadPayRoll(VAR P:TEXT);
VAR
   I :INTEGER;
BEGIN
   WITH PayRec DO
      BEGIN
         READLN(P, ID);
         READLN(P, Name);
         READLN(P, Wages);
         WRITE(ID:3,'  ');
         WRITE(Name);
{ Fill the rest of the 20 places with blanks }
         FOR I := 1 TO 20-LENGTH(Name) DO
            WRITE(' ');
         WRITELN(' $',Wages:0:2)
      END;
END;
{ -------------- Main Program ---------------- }
BEGIN
   ASSIGN(MasterFile, 'EMPFILE.BIN');   RESET(MasterFile);
   ASSIGN(Payfile, 'PAYFILE.TXT');      REWRITE(PayFile);
   WHILE NOT EOF(MasterFile) DO
      BEGIN
         GetInfo(MasterFile);
         WRITE('Please enter hours worked for this pay period: ');
         READLN(HoursWorked);
```

```
            CalcWages(HoursWorked, Wages);
            FilePayRoll(MasterFile, PayFile, Wages)
         END;
      CLOSE(MasterFile);
      CLOSE(PayFile);
      RESET(PayFile);
      WRITELN('--------- PayRoll Summary --------- ');
      WRITELN('ID # ------- Name -------- Salary');
      WHILE NOT EOF(PayFile) DO
         ReadPayroll(PayFile);
      WRITELN('---------------------------------- ');
      CLOSE(PayFile);
      WRITELN('Press ENTER to continue..');
      READLN
   END.
```

APPENDING A FILE If you would like to add the information for a new employee to the file EMPFILE.BIN, you cannot run the program 9-8 again because it will erase the whole file. There is another way to do this.

Adding data to an existing file is called appending, as the new data are written to the end of a sequential file. In some implementations (including Turbo) the file can be opened for appending using the procedure APPEND, which takes the form:

APPEND(file-variable);

While the REWRITE procedure positions the file pointer at the beginning of the file, APPEND positions the file pointer at the end of the file, so any new data will be written there. If your implementation does not have the procedure APPEND, use the following technique to add new items to the file:

- Open the file EMPFILE.BIN for reading using RESET.
- Open a scratch file (e.g. NEWFILE.TMP) for writing using REWRITE.
- Copy each item from EMPFILE.BIN to NEWFILE.TMP, then accept the new data from the keyboard and write them to NEWFILE.TMP.
- Open NEWFILE.TMP for reading and EMPFILE.BIN for writing, then copy the contents of NEWFILE.TMP back to EMPFILE.BIN.
- Erase the scratch file NEWFILE.TMP.

In standard Pascal, if the file variable is not included in the program header, the file is considered a temporary file and will be erased right after the program execution. You get the same result in UCSD if you close the file using the keyword PURGE.

In Turbo you can erase a file after closing it by using the procedure ERASE, which takes the form:

ERASE(file-variable);

If the information in the file needs to be changed (as in the case of a salary increase for employees), you can use a similar algorithm to update a sequential file. With random access files you can easily add or update records, but this kind of file will not be covered in our three days.

DRILL 9-4

Write a program that puts all the file tools you have learned together in one menu, using the payroll application. The menu should contain the following options:

- Display the Employee File
- Display an Employee record
- Add a new Employee

The following Menu procedure may be used in this program:

```
{ -------------- Procedure Menu ------------------ }
PROCEDURE Menu;
VAR
   Option :INTEGER;
BEGIN
   WRITELN(Header);
   WRITELN;
   WRITELN('1. Display employee file.');
   WRITELN('2. Display an employee record.');
   WRITELN('3. Add a new employee.');
   WRITELN('4. Exit.');
   WRITELN(Separator);
   WRITE('Make a choice and press a number: ');
   READLN(Option);
   CASE Option OF
      1 : Readit(DbFile);
      2 : ReadRec(DbFile, EmployeeRec);
      3 : AddRec(NewFile, DbFile, EmployeeRec);
      4 : Exit
   END;
   Menu
END;
```

As you can see in the Menu procedure, the options "1" to "3" correspond to the procedures you have to design. For the fourth option you may use the Turbo Pascal procedure (EXIT), a GOTO, or any suitable statement in your compiler that lets you exit from the repeated menu. Notice also that in this example a scratch file "NewFile" was used for adding a new employee to the file (option #3), but if you have the procedure APPEND in your compiler, you should use it instead, as it will save you a lot of effort. This program is the nucleus of a database and can be modified to include more features, such as updating employees' information and removing unwanted records from the database.

SUMMARY

In this chapter you learned the main tools for handling data files:

1. You know that standard Pascal provided TEXT and non-TEXT sequential files, while modern versions also provide random/direct access files.
2. During your tour of sequential files, you learned how to declare, create, write to, and read from a file.
3. TEXT files are declared using the form:

   file-variable **:TEXT;**

4. Files of other types are declared using the forms:

   TYPE
   type-identifier = **FILE OF** component-type;
   VAR
   file-variable :type-identifier;

   In standard Pascal the file-variable must be included as one of the file parameters or the file will be considered a temporary one and automatically deleted after the execution. In UCSD you must use the PURGE keyword to delete such a temporary file, and in Turbo the ERASE procedure.
5. The procedures used to open a file for either input or output are:

   RESET(file-variable); (for input)
   REWRITE(file-variable); (for output)

   With modern versions of Pascal (such as Turbo) you can also open a sequential file for appending with the procedure APPEND, which has a similar form to those above.
6. With versions other than standard Pascal, the file-variable must be linked to the actual file name on the disk. In Turbo this is done by using the

procedure ASSIGN; in UCSD the actual file name comes as a second parameter of the RESET or REWRITE procedures. In these implementations the file must be closed after processing using the CLOSE procedure.

7. You learned the general form of the input/output statements:

   READLN(file-variable, input-list);
   READ(file-variable, input-list);
   WRITELN(file-variable, output-list);
   WRITE(file-variable, output-list);

   and you know that the READLN and WRITELN procedures may not be used with non-TEXT files.

8. You also learned the general form of EOF and EOLN functions:

   EOF(file-variable)
   EOLN(file-variable)

Finally, you have had enough practice to enable you to create and manipulate files for different applications.

THE NEXT STEP

By the end of the third day you have enough tools to practice programming in Pascal and create good application programs. However, you may want to read about the main topics which were not covered in this book, which are:

- Variant Records
- Pointers and Linked Lists
- Direct/Random access files
- Graphics

The last two topics are not included in standard Pascal, so you have to refer to books on your specific compiler.

Other books on Turbo Pascal by Wordware Publishing, Inc. include *Graphic Programming with Turbo Pascal* and *Illustrated Turbo Pascal 6.0*.

APPENDIXES

APPENDIX A

THE ASCII CHARACTER SET

1. The Printable Characters

Decimal	Octal	Hexadecimal	Character	Decimal	Octal	Hexadecimal	Character
32	40	20	space	58	72	3a	:
33	41	21	!	59	73	3b	;
34	42	22	"	60	74	3c	
35	43	23	#	61	75	3d	=
36	44	24	$	62	76	3e	
37	45	25	%	63	77	3f	?
38	46	26	&	64	100	40	@
39	47	27	'	65	101	41	A
40	50	28	(	66	102	42	B
41	51	29	)	67	103	43	C
42	52	2a	*	68	104	44	D
43	53	2b	+	69	105	45	E
44	54	2c	,	70	106	46	F
45	55	2d	-	71	107	47	G
46	56	2e	.	72	110	48	H
47	57	2f	/	73	111	49	I
48	60	30	0	74	112	4a	J
49	61	31	1	75	113	4b	K
50	62	32	2	76	114	4c	L
51	63	33	3	77	115	4d	M
52	64	34	4	78	116	4e	N
53	65	35	5	79	117	4f	O
54	66	36	6	80	120	50	P
55	67	37	7	81	121	51	Q
56	70	38	8	82	122	52	R
57	71	39	9	83	123	53	S

1. The Printable Characters (cont.)

Decimal	Octal	Hexadecimal	Character	Decimal	Octal	Hexadecimal	Character
84	124	54	T	106	152	6a	j
85	125	55	U	107	153	6b	k
86	126	56	V	108	154	6c	l
87	127	57	W	109	155	6d	m
88	130	58	X	110	156	6e	n
89	131	59	Y	111	157	6f	o
90	132	5a	Z	112	160	70	p
91	133	5b	[	113	161	71	q
92	134	5c	\	114	162	72	r
93	135	5d	]	115	163	73	s
94	136	5e	^	116	164	74	t
95	137	5f	_	117	165	75	u
96	140	60	'	118	166	76	v
97	141	61	a	119	167	77	w
98	142	62	b	120	170	78	x
99	143	63	c	121	171	79	y
100	144	64	d	122	172	7a	z
101	145	65	e	123	173	7b	{
102	146	66	f	124	174	7c	|
103	147	67	g	125	175	7d	}
104	150	68	h	126	176	7e	~
105	151	69	i				

2. The Control Characters

Decimal	Octal	Hexadecimal	Key	Mnemonic Code
0	0	0	^@	NUL
1	1	1	^A	SOH
2	2	2	^B	STX
3	3	3	^C	ETX
4	4	4	^D	EOT
5	5	5	^E	ENQ
6	6	6	^F	ACK
7	7	7	^G	BEL
8	10	8	^H	BS
9	11	9	^I	HT
10	12	a	^J	LF
11	13	b	^K	VT
12	14	c	^L	FF
13	15	d	^M	CR
14	16	e	^N	SO
15	17	f	^O	SI
16	20	10	^P	DLE
17	21	11	^Q	DC1
18	22	12	^R	DC2
19	23	13	^S	DC3
20	24	14	^T	DC4
21	25	15	^U	NAK
22	26	16	^V	SYN
23	27	17	^W	ETB
24	30	18	^X	CAN
25	31	19	^Y	EM
26	32	1a	^Z	SUB
27	33	1b	ESC	ESC
28	34	1c	FS	
29	35	1d	GS	
30	36	1e	RS	
31	37	1f	US	
127	177	7f	DEL	DEL

APPENDIX B

RESERVED WORDS AND STANDARD IDENTIFIERS

1. RESERVED WORDS

AND	FORWARD	PROCEDURE
ARRAY	FUNCTION	PROGRAM
BEGIN	GOTO	RECORD
CASE	IF	REPEAT
CONST	IN	SET
DIV	LABEL	THEN
DO	MOD	TO
DOWNTO	NIL	TYPE
ELSE	NOT	UNTIL
END	OF	VAR
FILE	OR	WHILE
FOR	PACKED	WITH

Additional words reserved in Turbo Pascal:

ABSOLUTE	INTERFACE	STRING
EXTERNAL	INTERRUPT	UNIT
IMPLEMENTATION	SHL	USES
INLINE	SHR	XOR

2. STANDARD IDENTIFIERS

CONSTANTS

FALSE	MAXINT	TRUE

TYPES

BOOLEAN	CHAR	INTEGER
REAL	TEXT	

FILES

INPUT	OUTPUT	

FUNCTIONS

ABS	ARCTAN	CHR
COS	EOF	EOLN
EXP	LN	ODD
ORD	PRED	ROUND
SIN	SQR	SQRT
SUCC	TRUNC	

PROCEDURES

DISPOSE	GET	NEW
PACK	PAGE	PUT
READ	READLN	RESET
REWRITE	UNPACK	WRITE
WRITELN		

Additional identifiers predefined in Turbo Pascal

CONSTANTS

MAXLONGINT	PI	

TYPES

BYTE	COMP	DOUBLE
EXTENDED	LONGINT	SHORTINT
SINGLE	WORD	

FUNCTIONS (discussed in this book)

CONCAT	COPY	LENGTH
PI	POS	RANDOM

PROCEDURES (discussed in this book)

APPEND	ASSIGN	CLOSE
DELETE	EXIT	INSERT

NOTE: The standard procedures GET, PUT, PACK, UNPACK, and PAGE are not defined in Turbo Pascal.

Index

4c/=

POPULAR APPLICATIONS SERIES

LEARN Pascal IN THREE DAYS

Designed to show beginning programmers the basics of programming in Pascal, this book is broken down into specific objectives organized into Day 1, Day 2, and Day 3. Step-by-step instructions and hands-on activities guide readers through the basic programming structures and methods needed to effectively program using Pascal. Companion diskette with source code examples available

SAM ABOLROUS is a software engineer with an extensive background in software design and programming. Mr. Abolrous' published works include numerous articles for industry journals and over thirty books on computer applications ranging from DOS to COBOL programming including Wordware's *Learn C in Three Days*.

COMPANION DISK AVAILABLE

SEPARATELY FOR Rs. 75.00

INCLUSIVE OF PACKING & POSTAGE

ISBN 81-7029-268-9

BPB PUBLICATIONS

B-14, CONNAUGHT PLACE, NEW DELHI-110001

Rs. 39/-